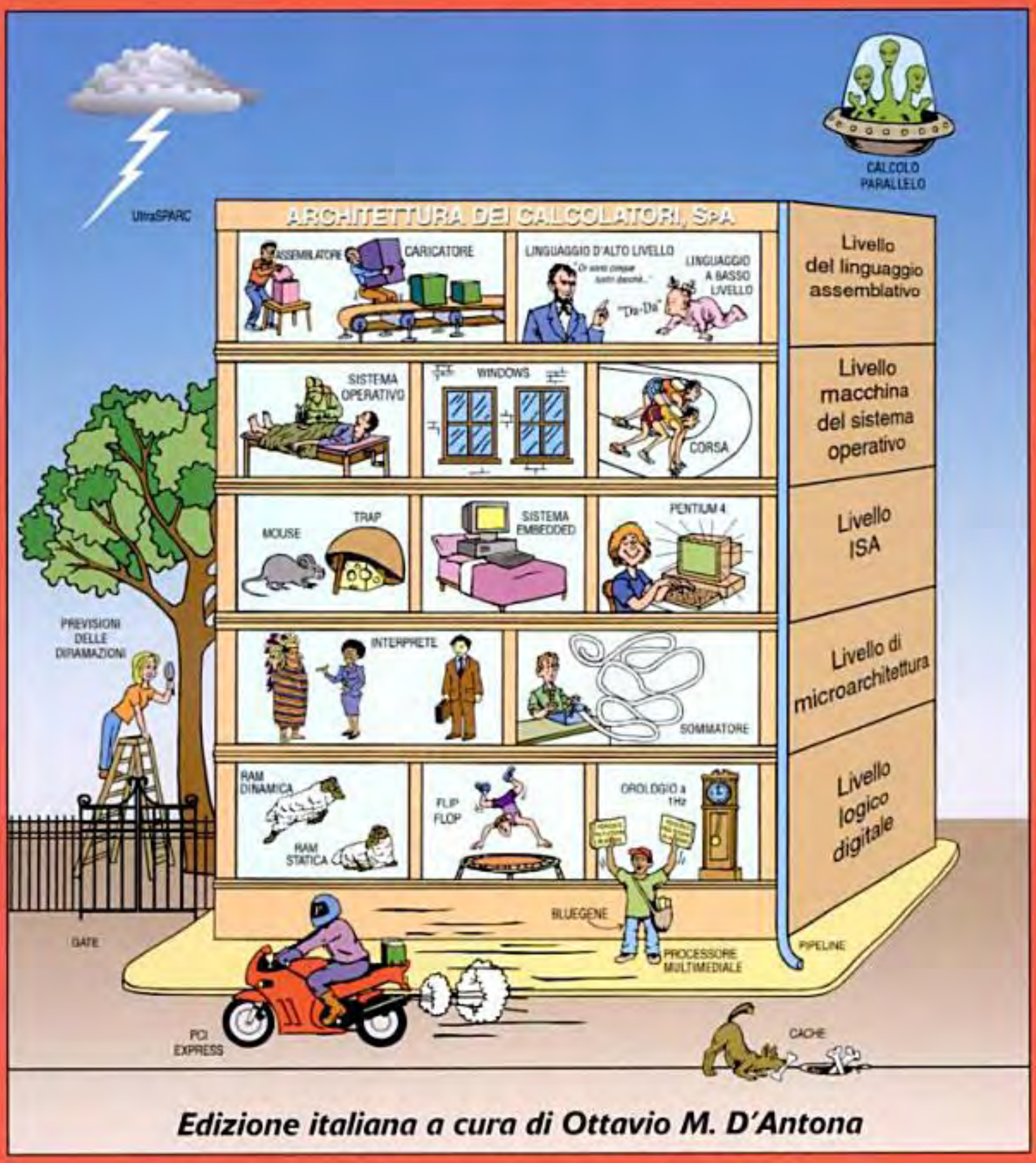
Andrew S. Tanenbaum

Architettura dei Calcolatori
un approccio strutturale



Edizione italiana a cura di Ottavio M. D'Antona

Quinta edizione

PEARSON
Prentice Hall

VISIT…

Andrew S. Tanenbaum

Architettura dei calcolatori

Un approccio strutturale

Quinta edizione

Edizione italiana a cura di Ottavio M. D'Antona

PEARSON
Prentice Hall

Copyright © 2006 Pearson Education Italia S.r.l.
Via Fara, 28 – 20124 Milano
Tel. 026739761 – Fax 02673976503
E-mail: hpeitalia@pearson.com
Web: http://hpe.pearsoned.it

Authorized translation from the English language edition, entitled *STRUCTURED COMPUTER ORGANIZATION*, 5th Edition by TANENBAUM, ANDREW S., published by Pearson Education, Inc., publishing as Prentice Hall, Copyright © 2006.
All Rights Reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

Italian language edition published by Pearson Education Italia S.r.l., Copyright © 2006

Le informazioni contenute in questo libro sono state verificate e documentate con la massima cura possibile. Nessuna responsabilità derivante dal loro utilizzo potrà venire imputata agli Autori, a Pearson Education Italia o a ogni persona e società coinvolta nella creazione, produzione e distribuzione di questo libro.

I diritti di riproduzione e di memorizzazione elettronica totale e parziale con qualsiasi mezzo, compresi i microfilm e le copie fotostatiche, sono riservati per tutti i paesi.
LA FOTOCOPIATURA DEI LIBRI È UN REATO. L'editore potrà concedere a pagamento l'autorizzazione a riprodurre una porzione non superiore a un decimo del presente volume. Le richieste di riproduzione vanno inoltrate ad AIDRO (Associazione Italiana per i Diritti di Riproduzione delle Opere dell'Ingegno), Via delle Erbe, 2 - 20121 Milano - Tel. e Fax 02.80.95.06.

Curatore per l'edizione italiana: Ottavio M. D'Antona

Traduzione: Giuseppe Lipori, Massimiliano Piscozzi
Copy-editing: Donatella Pepe
Composizione: Elisabetta Bozzi
Grafica di copertina: Sabrina Miraglia
Illustrazione di copertina: Maraska Artistic Concept and Design Don Martinetti, DM Graphics, Inc.
Stampa: Legoprint – Lavis (TN)

Tutti i marchi citati nel testo sono di proprietà dei loro detentori.

ISBN13: 978-8-8719-2271-3
ISBN10: 88-7192-271-9

Printed in Italy

5ª edizione: aprile 2006

A Suzanne, Barbara, Marvin e in memoria di Sweetie π e di Bram

Sommario

Prefazione all'edizione italiana xv

Prefazione xvii

Capitolo 1 Introduzione 1

1.1 Approccio strutturale 2
- 1.1.1 Linguaggi, livelli e macchine virtuali 2
- 1.1.2 Attuali macchine multilivello 4
- 1.1.3 Evoluzione delle macchine multilivello 7

1.2 Pietre miliari nell'architettura dei computer 12
- 1.2.1 Generazione zero – Computer meccanici (1642-1945) 13
- 1.2.2 Prima generazione – Valvole (1945-1955) 15
- 1.2.3 Seconda generazione – Transistor (1955-1965) 18
- 1.2.4 Terza generazione – Circuiti integrati (1965-1980) 20
- 1.2.5 Quarta generazione – Integrazione a grandissima scala (1980-?) 22
- 1.2.6 Quinta generazione – Computer invisibili 24

1.3 Tipologie di computer 26
- 1.3.1 Forze tecnologiche ed economiche 26
- 1.3.2 Spettro di computer 28
- 1.3.3 Computer usa e getta 28
- 1.3.4 Microcontrollori 30
- 1.3.5 Computer per giocare 32
- 1.3.6 Personal computer 33
- 1.3.7 Server 34
- 1.3.8 Cluster di workstation 34
- 1.3.9 Mainframe 35

1.4 Esempi di famiglie di computer 36
- 1.4.1 Introduzione al Pentium 4 36
- 1.4.2 Introduzione all'UltraSPARC III 41
- 1.4.3 Introduzione all'8051 42

1.5 Unità metriche 45

1.6 Organizzazione del libro 46

Problemi 47

Capitolo 2 Organizzazione dei sistemi di calcolo 49

2.1 Processori 49
2.1.1 Organizzazione della CPU 50
2.1.2 Esecuzione dell'istruzione 51
2.1.3 RISC contro CISC 55
2.1.4 Principi di progettazione dei calcolatori moderni 57
2.1.5 Parallelismo a livello d'istruzione 58
2.1.6 Parallelismo a livello di processore 62
2.2 Memoria principale 66
2.2.1 Bit 66
2.2.2 Indirizzi di memoria 67
2.2.3 Ordinamento dei byte 68
2.2.4 Codici correttori 70
2.2.5 Memoria cache 74
2.2.6 Assemblaggio e tipi di memoria 77
2.3 Memoria secondaria 78
2.3.1 Gerarchie di memoria 78
2.3.2 Dischi magnetici 79
2.3.3 Floppy disk 83
2.3.4 Dischi IDE 83
2.3.5 Dischi SCSI 85
2.3.6 RAID 86
2.3.7 CD-ROM 90
2.3.8 CD-registrabili 93
2.3.9 CD-riscrivibili 96
2.3.10 DVD 96
2.3.11 Blu-Ray 98
2.4 Input/Output 99
2.4.1 Bus 99
2.4.2 Terminali 102
2.4.3 Mouse 106
2.4.4 Stampanti 108
2.4.5 Apparecchiature per le telecomunicazioni 113
2.4.6 Macchine fotografiche digitali 121
2.4.7 Codifica dei caratteri 123
2.5 Riepilogo 127
Problemi 128

Capitolo 3 Livello logico digitale 131

3.1 Porte logiche e algebra di Boole 131
3.1.1 Porte logiche 131
3.1.2 Algebra di Boole 134
3.1.3 Implementazione delle funzioni booleane 136
3.1.4 Equivalenza di circuiti 138
3.2 Circuiti logici digitali elementari 141
3.2.1 Circuiti integrati 141
3.2.2 Reti combinatorie 143
3.2.3 Circuiti per l'aritmetica 148
3.2.4 Clock 152

3.3 Memoria 154
3.3.1 Latch 154
3.3.2 Flip-flop 157
3.3.3 Registri 159
3.3.4 Organizzazione della memoria 160
3.3.5 Chip di memoria 163
3.3.6 RAM e ROM 166
3.4 Chip della CPU e bus 168
3.4.1 Chip della CPU 168
3.4.2 Bus del calcolatore 170
3.4.3 Ampiezza del bus 173
3.4.4 Temporizzazione del bus 174
3.4.5 Arbitraggio del bus 179
3.4.6 Operazioni del bus 181
3.5 Esempi di chip della CPU 184
3.5.1 Pentium 4 184
3.5.2 UltraSPARC III 190
3.5.3 Chip 8051 194
3.6 Esempi di bus 196
3.6.1 Bus ISA 197
3.6.2 Bus PCI 198
3.6.3 PCI Express 206
3.6.4 Universal Serial Bus 210
3.7 Interfacce 214
3.7.1 Chip di I/O 214
3.7.2 Decodifica dell'indirizzo 215
3.8 Riepilogo 218
Problemi 219

Capitolo 4 Livello di microarchitettura 223

4.1 Esempio di microarchitettura 223
4.1.1 Percorso dati 224
4.1.2 Microistruzioni 230
4.1.3 Unità di controllo microprogrammata: Mic-1 233
4.2 Esempio di ISA: IJVM 238
4.2.1 Stack 238
4.2.2 Modello della memoria di IJVM 240
4.2.3 Insieme d'istruzioni IJVM 242
4.2.4 Compilazione da Java a IJVM 246
4.3 Implementazione di esempio 247
4.3.1 Microistruzioni e notazione 247
4.3.2 Implementazione di IJVM con Mic-1 251
4.4 Progettazione del livello di microarchitettura 263
4.4.1 Velocità/costi 263
4.4.2 Riduzione della lunghezza del percorso di esecuzione 266
4.4.3 Architettura con prefetching: Mic-2 272
4.4.4 Architettura a pipeline: Mic-3 276
4.4.5 Pipeline a sette stadi: Mic-4 281

4.5 Miglioramento delle prestazioni 284
4.5.1 Memoria cache 285
4.5.2 Predizione dei salti 291
4.5.3 Esecuzione fuori sequenza e rinomina dei registri 296
4.5.4 Esecuzione speculativa 301
4.6 Esempi del livello di microarchitettura 304
4.6.1 Microarchitettura della CPU Pentium 4 304
4.6.2 Microarchitettura della CPU UltraSPARC III Cu 310
4.6.3 Microarchitettura della CPU 8051 315
4.7 Confronto tra Pentium, UltraSPARC e 8051 318
4.8 Riepilogo 319
Problemi 320

Capitolo 5 Livello di architettura 323

5.1 Panoramica del livello ISA 325
5.1.1 Proprietà del livello ISA 325
5.1.2 Modelli di memoria 327
5.1.3 Registri 329
5.1.4 Istruzioni 330
5.1.5 Panoramica del livello ISA del Pentium 4 330
5.1.6 Panoramica del livello ISA dell'UltraSPARC III 333
5.1.7 Panoramica del livello ISA dell'8051 336
5.2 Tipi di dati 339
5.2.1 Tipi di dati numerici 339
5.2.2 Tipi di dati non numerici 340
5.2.3 Tipi di dati del Pentium 4 341
5.2.4 Tipi di dati dell'UltraSPARC III 341
5.2.5 Tipi di dati dell'8051 342
5.3 Formati d'istruzione 342
5.3.1 Criteri progettuali per i formati d'istruzioni 343
5.3.2 Codice operativo espandibile 345
5.3.3 Formati delle istruzioni di Pentium 4 348
5.3.4 Formati delle istruzioni di UltraSPARC III 349
5.3.5 Formati delle istruzioni di 8051 350
5.4 Indirizzamento 351
5.4.1 Modalità d'indirizzamento 351
5.4.2 Indirizzamento immediato 351
5.4.3 Indirizzamento diretto 352
5.4.4 Indirizzamento a registro 352
5.4.5 Indirizzamento a registro indiretto 352
5.4.6 Indirizzamento indicizzato 353
5.4.7 Indirizzamento indicizzato esteso 355
5.4.8 Indirizzamento a stack 355
5.4.9 Modalità d'indirizzamento per istruzioni di salto 359
5.4.10 Ortogonalità dei codici operativi e delle modalità d'indirizzamento 359
5.4.11 Modalità d'indirizzamento del Pentium 4 361
5.4.12 Modalità d'indirizzamento di UltraSPARC III 363
5.4.13 Modalità d'indirizzamento di 8051 363
5.4.14 Analisi delle modalità d'indirizzamento 364

5.5 Tipi d'istruzioni 365
5.5.1 Istruzioni di trasferimento dati 365
5.5.2 Operazioni binarie 366
5.5.3 Operazioni unarie 367
5.5.4 Confronti e salti condizionati 369
5.5.5 Invocazione di procedura 370
5.5.6 Istruzioni di ciclo 371
5.5.7 Input/Output 372
5.5.8 Istruzioni di Pentium 4 375
5.5.9 Istruzioni di UltraSPARC III 378
5.5.10 Istruzioni di 8051 381
5.5.11 Insiemi d'istruzioni a confronto 381
5.6 Controllo del flusso 384
5.6.1 Flusso sequenziale e diramazioni 384
5.6.2 Procedure 385
5.6.3 Coroutine 390
5.6.4 Trap 392
5.6.5 Interrupt 393
5.7 Un esempio: le torri di Hanoi 396
5.7.1 Le torri di Hanoi nel linguaggio assemblativo di Pentium 4 397
5.7.2 Le torri di Hanoi nel linguaggio assemblativo di UltraSPARC III 397
5.8 Architettura IA-64 e Itanium 2 400
5.8.1 Il problema di Pentium 4 401
5.8.2 Modello IA-64 e calcolo che utilizza il parallelismo esplicito 402
5.8.3 Riduzione degli accessi in memoria 403
5.8.4 Scheduling delle istruzioni 404
5.8.5 Riduzione dei salti condizionati: attribuzione di predicati 405
5.8.6 Caricamenti speculativi 407
5.9 Riepilogo 408
Problemi 409

Capitolo 6 Livello macchina del sistema operativo 415

6.1 Memoria virtuale 416
6.1.1 Paginazione 417
6.1.2 Implementazione della paginazione 419
6.1.3 Paginazione a richiesta e working set 422
6.1.4 Politica di sostituzione delle pagine 423
6.1.5 Dimensione di pagina e frammentazione 425
6.1.6 Segmentazione 426
6.1.7 Implementazione della segmentazione 429
6.1.8 Memoria virtuale del Pentium 4 432
6.1.9 Memoria virtuale dell'UltraSPARC III 436
6.1.10 Memoria virtuale e caching 439
6.2 Istruzioni virtuali di I/O 440
6.2.1 File 440
6.2.2 Implementazione delle istruzioni virtuali di I/O 442
6.2.3 Istruzioni per la gestione di directory 445

6.3 Istruzioni virtuali per il calcolo parallelo 446
6.3.1 Creazione dei processi 447
6.3.2 Corsa critica 448
6.3.3 Sincronizzazione dei processi tramite semafori 452
6.4 Sistemi operativi di esempio 455
6.4.1 Introduzione 455
6.4.2 Esempi di memoria virtuale 463
6.4.3 Esempi di I/O virtuale 467
6.4.4 Esempi di gestione dei processi 477
6.5 Riepilogo 483
Problemi 485

Capitolo 7 Livello del linguaggio assemblativo 491
7.1 Introduzione al linguaggio assemblativo 492
7.1.1 Che cos'è un linguaggio assemblativo 492
7.1.2 Perché usare il linguaggio assemblativo 493
7.1.3 Formato delle istruzioni del linguaggio assemblativo 495
7.1.4 Pseudoistruzioni 498
7.2 Macroistruzioni 501
7.2.1 Definizione, chiamata ed espansione di macro 501
7.2.2 Macro con parametri 503
7.2.3 Caratteristiche avanzate 504
7.2.4 Implementazione delle funzionalità macro negli assemblatori 504
7.3 Processo di assemblaggio 505
7.3.1 Assemblatori a due passate 505
7.3.2 Prima passata 506
7.3.3 Seconda passata 510
7.3.4 Tabella dei simboli 511
7.4 Collegamento e caricamento 513
7.4.1 Compiti del linker 514
7.4.2 Struttura di un modulo oggetto 517
7.4.3 Rilocazione a tempo del binding e dinamica 518
7.4.4 Collegamento dinamico 521
7.5 Riepilogo 525
Problemi 526

Capitolo 8 Architetture per il calcolo parallelo 529
8.1 Parallelismo nel chip 531
8.1.1 Parallelismo a livello delle istruzioni 531
8.1.2 Multithreading nel chip 537
8.1.3 Multiprocessori in un solo chip 543
8.2 Coprocessori 548
8.2.1 Processori di rete 548
8.2.2 Processori multimediali 556
8.2.3 Crittoprocessori 561

8.3 Multiprocessori con memoria condivisa 562
8.3.1 Multiprocessori e multicomputer a confronto 562
8.3.2 Semantica della memoria 569
8.3.3 Architetture di multiprocessori simmetrici UMA 573
8.3.4 Multiprocessori NUMA 581
8.3.5 Multiprocessori COMA 589
8.4 Multicomputer a scambio di messaggi 591
8.4.1 Reti d'interconnessione 592
8.4.2 MPP: processori massicciamente paralleli 595
8.4.3 Cluster 604
8.4.4 Software di comunicazione per multicomputer 609
8.4.5 Scheduling 612
8.4.6 Memoria condivisa a livello applicativo 613
8.4.7 Prestazioni 620
8.5 Grid computing 626
8.6 Riepilogo 629
Problemi 630

Capitolo 9 Bibliografia e letture consigliate 633
9.1 Letture di approfondimento 633
9.1.1 Lavori introduttivi 633
9.1.2 Organizzazione dei calcolatori 635
9.1.3 Livello logico digitale 636
9.1.4 Livello di microarchitettura 636
9.1.5 Livello di architettura dell'insieme d'istruzioni 637
9.1.6 Livello macchina del sistema operativo 637
9.1.7 Livello del linguaggio assemblativo 638
9.1.8 Architetture per il calcolo parallelo 638
9.1.9 Aritmetica binaria 640
9.1.10 Programmazione in linguaggio assemblativo 641
9.2 Bibliografia alfabetica 641

Appendice A Aritmetica binaria 655
A.1 Numeri a precisione finita 655
A.2 Sistemi di numerazione in base fissa 657
A.3 Conversione tra basi 658
A.4 Numeri binari negativi 662
A.5 Aritmetica binaria 664
Problemi 665

Appendice B Numeri in virgola mobile 667
B.1 Principi dell'aritmetica in virgola mobile 668
B.2 Standard in virgola mobile IEEE 754 671
Problemi 675

Appendice C Programmazione in linguaggio assemblativo 677

C.1 Panoramica 678
C.1.1 Linguaggio assemblativo 678
C.1.2 Breve programma in linguaggio assemblativo 678
C.2 Processore 8088 680
C.2.1 Ciclo del processore 680
C.2.2 Registri d'uso generale 681
C.2.3 Registri puntatore 683
C.3 Memoria e indirizzamento 684
C.3.1 Organizzazione di memoria e segmenti 684
C.3.2 Indirizzamento 686
C.4 Istruzioni dell'8088 689
C.4.1 Trasferimento, copia e aritmetica 689
C.4.2 Operazioni logiche, su bit e di scorrimento 692
C.4.3 Cicli e operazioni iterative su stringhe 692
C.4.4 Istruzioni di salto e di chiamata 693
C.4.5 Chiamate di subroutine 695
C.4.6 Chiamate di sistema e subroutine di sistema 696
C.4.7 Osservazioni sull'insieme d'istruzioni 698
C.5 Assemblatore 699
C.5.1 Introduzione 699
C.5.2 as88, un assemblatore basato su ACK 700
C.5.3 Differenze tra assemblatori dell'8088 703
C.6 Tracer 705
C.6.1 Comandi del tracer 706
C.7 Installazione 708
C.8 Esempi 709
C.8.1 Esempio Hello World 709
C.8.2 Esempio con i registri d'uso generale 712
C.8.3 Istruzioni di chiamata e puntatori ai registri 713
C.8.4 Debugging di un programma per la stampa di array 716
C.8.5 Manipolazione di stringhe e istruzioni su stringhe 719
C.8.6 Tabella di salto 722
C.8.7 File: accesso diretto e accesso bufferizzato 723
Problemi 727

Indice analitico 729

Prefazione all'edizione italiana

Il passato, il presente e il futuro prossimo, anzi probabile, delle architetture dei calcolatori: ecco, in estrema sintesi, il contenuto del libro di Tanenbaum. L'autore, dopo aver visitato le innumerevoli pieghe dell'articolata materia, offre al lettore, con generosità e chiarezza, il risultato dell'indagine svolta. Parlare di completezza della trattazione in un'opera scientifica è sempre rischioso, ma anche il più prudente riconoscerà che, mai come questa volta, Tanenbaum ci è andato molto vicino. Questo è un libro che serve per imparare!

Tra i punti che vale la pena segnalare, penso alla meticolosa descrizione delle stampanti laser, alle varie forme di parallelismo, alla presentazione delle innovazioni esecutive come l'elaborazione speculativa e, in particolar modo, all'attenta illustrazione dei concetti di base e dello spirito della programmazione a livello macchina. Un tema di cui, per quanto arduo e spigoloso, non c'è didattica che possa fare a meno. E infatti il testo è accompagnato da un CD-ROM contenente un bel simulatore dell'Intel 8088 (ampiamente descritto nell'Appendice C) con cui si possono scrivere programmi macchina eseguibili dai Pentium.

Infine, mi piace segnalare che tra tanto tecnicismo le "battutine alla Tanenbaum" di cui è punteggiato il libro sono un vero e proprio... *breath of fresh air.*

Prof. Ottavio M. D'Antona
Dipartimento di Informatica e Comunicazione
Università degli Studi di Milano

Prefazione

Le prime quattro edizioni di questo libro erano basate sull'idea che un computer può essere visto come una gerarchia di livelli, in cui ciascuno realizza una particolare e ben definita funzione. Questo fondamentale concetto è valido ancor oggi così come lo era quando venne pubblicata la prima edizione; per questo motivo è stato mantenuto come base anche di questa pubblicazione. Come nelle quattro precedenti edizioni, sono trattati in modo dettagliato il livello logico digitale, il livello di microarchitettura, l'insieme delle istruzioni, il livello di architettura, il livello macchina del sistema operativo e il livello del linguaggio assemblativo.

Pur mantenendo quindi la struttura di base, questa quinta edizione contiene molti cambiamenti, che permettono al testo di restare al passo con la rapida evoluzione dell'industria dei computer. Sono state aggiornate anche le macchine utilizzate come casi di studio nel testo; gli esempi presentati sono ora l'Intel Pentium 4, il Sun Ultra-SPARC III e l'Intel 8051. Il Pentium 4 è un esempio di CPU utilizzata largamente su macchine desktop, mentre l'UltraSPARC III è un esempio di server molto diffuso, ampiamente impiegato in sistemi multiprocessore di medie e grandi dimensioni.

La scelta del processore 8051 potrebbe tuttavia risultare a qualcuno inaspettata. Si tratta di un chip di vecchia data in circolazione da decenni, tornato infine in auge grazie all'enorme crescita dei sistemi integrati. Dato che ormai i computer vengono impiegati ovunque, dalle radiosveglie digitali ai forni a microonde, l'interesse per i sistemi integrati sta crescendo rapidamente. L'8051 rappresenta una scelta ampiamente diffusa grazie al costo estremamente basso (nell'ordine dei centesimi di euro), alla grande disponibilità di software e periferiche e all'esistenza di un gran numero di programmatori specializzati.

Nel corso degli anni, molti professori mi hanno richiesto a più riprese materiale sulla programmazione in linguaggio assemblativo. Con la quinta edizione questo materiale è finalmente disponibile ed è possibile trovarlo sia nell'Appendice C sia nel CD-ROM allegato. Il linguaggio assemblativo scelto è quello dell'8088, dato che rappresenta una versione molto ridotta del Pentium, un processore assai diffuso. Avrei potuto utilizzare l'UltraSPARC, il MIPS o una qualche altra CPU della quale quasi nessuno ha mai sentito parlare, ma l'8088, come strumento di motivazione, rappresenta la scelta migliore dato che molti studenti possiedono un Pentium a casa e dato che su di esso è possibile eseguire programmi 8088. Tuttavia, dato che effettuare il debug del codice assemblativo è particolarmente difficile, ho messo a disposizione un insieme di strumenti per facilitare l'apprendimento della programmazione

assemblativa, tra cui un assemblatore 8088, un simulatore e un *tracer*. Questi strumenti sono forniti per Windows, Unix e Linux e si trovano sia nel CD-ROM sia sul sito Internet del libro (citato in seguito).

Nel corso degli anni il libro si è "allungato". Una crescita inevitabile in quanto la materia continua a svilupparsi e vi è un numero sempre maggiore di cose da conoscere. Di conseguenza, quando il libro è utilizzato come testo di un corso accademico semestrale potrebbe non essere possibile terminarlo nell'arco delle lezioni. Un possibile approccio potrebbe essere quello di studiare, come minimo indispensabile, i Capitoli 1, 2 e 3 interamente, la prima parte del Capitolo 4 (fino al Paragrafo 4.4 compreso) e il Capitolo 5. L'eventuale tempo rimanente potrebbe essere impiegato per il resto del Capitolo 4 e per parti dei Capitoli 6, 7 e 8 selezionate in base alle preferenze del docente.

Vediamo ora rapidamente, capitolo per capitolo, le principali novità di questa quinta edizione. Il Capitolo 1 presenta ancora una panoramica storica delle architetture dei computer, nella quale si evidenzia come si sia raggiunto lo stato attuale presentando le pietre miliari che hanno marcato questo cammino. Viene trattata una più ampia gamma di computer e sono introdotti i tre principali esempi usati nel testo (Pentium 4, UltraSPARC III e 8051).

Nel Capitolo 2 sono stati aggiornati gli argomenti relativi all'input/output, prestando maggior attenzione alle tecnologie dei moderni dispositivi, tra cui le macchine fotografiche digitali, le DSL e Internet via cavo.

Il Capitolo 3 è stato sottoposto a un'intensa revisione, e ora tratta i bus dei computer e i moderni chip di I/O. In questo capitolo vengono descritti i tre nuovi esempi a livello di processore. È stato aggiunto materiale relativo al bus PCI Express, che ci si aspetta sostituirà a breve il bus PCI.

Il Capitolo 4 è sempre stato molto apprezzato per la chiara spiegazione del reale funzionamento di un computer; per questo motivo non ha subìto modifiche rispetto alla precedente edizione. Sono stati comunque aggiunti alcuni nuovi paragrafi che trattano il livello di microarchitettura del Pentium 4, dell'UltraSPARC III e dell'8051.

I Capitoli 5, 6 e 7 sono stati aggiornati usando nuovi esempi, ma al di là di questa modifica sono rimasti sostanzialmente immutati. Nonostante il Capitolo 6 usi ora come esempio Windows XP al posto di Windows NT, i cambiamenti nella trattazione degli argomenti sono minimi.

Al contrario, il Capitolo 8 ha subìto profonde modifiche per restare al passo con lo stato attuale delle diverse tipologie di computer paralleli. Tratta cinque distinte classi di sistemi paralleli, dal parallelismo nel chip (parallelismo a livello d'istruzione, multithreading nel chip, multiprocessori a singolo chip), ai coprocessori, ai sistemi a memoria condivisa e ai cluster, per terminare infine con una breve analisi dei grid. Sono inoltre presentati vari nuovi esempi, tra cui CPU TriMedia, BlueGene/L, Red Storm e i cluster di Google.

Dato che l'architettura dei calcolatori è un campo in continua evoluzione, molti dei riferimenti citati nel Capitolo 9 sono stati aggiornati. Oltre la metà di quelli presenti in questa quinta edizione sono libri e articoli scientifici scritti successivamente alla quarta pubblicazione di questo testo.

Le Appendici A e B non sono state modificate rispetto all'edizione precedente, mentre l'Appendice C sulla programmazione assemblativa è completamente nuova. È una guida pratica e facile da consultare sulla programmazione in linguaggio assemblativo attraverso gli strumenti forniti nel CD-ROM. L'Appendice C è stata scritta dal Dott. Evert Wattel della Vrije Universiteit di Amsterdam. Il Dott. Wattel ha un'esperienza pluriennale d'insegnamento mediante questi strumenti; a lui vanno i miei ringraziamenti per aver scritto questa appendice.

Sul CD-ROM, oltre agli strumenti per il linguaggio assemblativo, si trova un simulatore grafico da utilizzare come ausilio nello studio del Capitolo 4. Questo simulatore è stato scritto dal Prof. Richard Salter dell'Oberlin College e può essere usato dagli studenti per comprendere meglio gli argomenti trattati in questo capitolo. Ringrazio il Prof. Richard Salter per aver scritto questo programma.

Sul sito web sono disponibili le presentazioni in PowerPoint per i docenti. Per poter raggiungere il sito web del libro occorre andare all'indirizzo

http:/www.prenhall.com/tanenbaum

e selezionare l'icona relativa a questo testo; un menu permette poi di scegliere le sezioni desiderate[1].

Varie persone hanno letto, interamente o in parte, il testo e hanno fornito utili suggerimenti o sono state d'aiuto in altro modo. In particolare vorrei ringraziare per il loro aiuto Nikitas Alexandridis, Shekar Borkar, Herbert Bos, Scott Cannon, Doug Carmean, Alan Charlesworth, Eric Cota-Robles, Michael Fetterman, Quinn Jacobson, Thilo Kielmann, Iffat Kazi, Saul Levy, Ahmed Louri, Abhijit Pandya, Krist Petersen, Mark Russinovich, Ronald Schroeder e Saim Ural. Sono loro riconoscente. Grazie.

Vorrei inoltre ringraziare Jim Goodman per il contributo che ha dato a questo libro, in particolare nei Capitoli 4 e 5. L'idea di usare la Java Virtual Machine è stata sua e il libro, grazie a questa scelta, ne ha tratto indubbio vantaggio.

Infine vorrei ringraziare ancora una volta Suzanne per il suo amore e la sua pazienza che, dopo 15 libri, sembra ormai infinita. Barbara e Marvin sono da sempre fonte di felicità e ora sono a conoscenza di che cosa fanno i professori per guadagnarsi da vivere. Nel 2004 la Royal Netherlands Academy of Arts and Sciences mi ha assegnato un premio molto ambito, l'Academy Professorship, liberandomi da alcuni degli aspetti meno attraenti della vita accademica (come le noiose e interminabili riunioni di comitato); di questo sarò loro eternamente grato.

Andrew S. Tanenbaum

---

[1] *I docenti che adottano questo testo in un corso universitario potranno richiedere le slide in italiano e il Solutions* Manual (in inglese) direttamente ai rappresentanti accademici Pearson Education Italia oppure inviando un messaggio e-mail a: hpeitalia@pearson.com (*N.d.E.*).

Capitolo 1

Introduzione

Un calcolatore digitale è una macchina che può risolvere problemi eseguendo le istruzioni che le vengono assegnate. Con il termine **programma** si intende una sequenza d'istruzioni che descrive come portare a termine un dato compito. I circuiti elettronici di un qualsiasi computer possono riconoscere ed eseguire direttamente soltanto un insieme limitato d'istruzioni semplici in cui tutti i programmi devono essere convertiti prima di poter essere eseguiti. Raramente queste istruzioni elementari sono molto più complicate delle seguenti:

- sommare due numeri
- controllare se un numero vale zero
- copiare una porzione di dati da una parte all'altra della memoria.

L'insieme di queste istruzioni primitive forma un linguaggio, chiamato **linguaggio macchina**, attraverso il quale è possibile comunicare con il computer. Chi progetta un calcolatore deve decidere quali istruzioni costituiranno il suo linguaggio macchina. Per ridurre la complessità e il costo dei dispositivi elettronici generalmente si cerca di progettare le più semplici istruzioni primitive che siano compatibili con il tipo di utilizzo e i requisiti prestazionali in base ai quali si progetta il computer. Dato che quasi tutti i linguaggi macchina sono semplici ed elementari, risulta difficile e tedioso utilizzarli.

Nel corso del tempo, questa semplice osservazione ha portato a strutturare i computer come una serie di livelli di astrazione, ciascuno dei quali è costruito sulla base di quello sottostante. In questo modo la complessità è gestibile più agevolmente e i computer possono essere progettati in modo sistematico e organizzato. Chiamiamo **approccio strutturale** questo modo di concepire l'architettura dei computer; nel paragrafo successivo, descriveremo in modo più preciso che cosa intendiamo con questo termine. Successivamente prenderemo in considerazione alcuni sviluppi storici, lo stato dell'arte e alcuni importanti esempi.

1.1 Approccio strutturale

Come abbiamo appena accennato esiste una gran differenza tra ciò che è adatto agli utenti e ciò che lo è per i computer. Gli utenti vogliono fare *X*, ma i computer possono fare soltanto *Y*: ciò genera un problema e l'obiettivo di questo libro è spiegare in che modo sia possibile risolverlo.

1.1.1 Linguaggi, livelli e macchine virtuali

Il problema può essere affrontato in due modi distinti: entrambi prevedono la definizione di un nuovo insieme d'istruzioni che sia più comodo da utilizzare rispetto a quanto non lo siano le istruzioni macchina predefinite. Considerate globalmente, anche queste nuove istruzioni formano un linguaggio, che chiamiamo L1, allo stesso modo in cui le istruzioni macchina formavano a loro volta un linguaggio, che chiamiamo L0. I due approcci differiscono nel modo in cui i programmi scritti in L1 possono essere eseguiti da un computer, in grado di eseguire soltanto quelli scritti nel proprio linguaggio macchina, L0.

Un metodo per eseguire un programma scritto in L1 consiste nel sostituire, in una fase iniziale, ogni sua istruzione con un'equivalente sequenza di istruzioni in L0. Il programma che ne risulta è costituito interamente da istruzioni di L0 e può essere eseguito dal computer al posto del programma L1 originale. Questa tecnica è chiamata **traduzione**.

L'altra tecnica consiste invece nello scrivere un programma in L0 che accetta come dati d'ingresso programmi in L1; tale programma li esegue esaminando un'istruzione alla volta e sostituendola direttamente con l'equivalente sequenza di istruzioni L0. Questa tecnica, che non richiede la generazione preventiva di un nuovo programma L0, è chiamata **interpretazione** e il programma che la esegue è detto **interprete**.

La traduzione e l'interpretazione sono simili. In entrambi i metodi il computer può trattare istruzioni in L1 eseguendo le equivalenti sequenze di istruzioni L0. La differenza è che, nel caso della traduzione, il programma L1 viene, all'inizio, convertito interamente in un programma L0. Il programma L1 può quindi essere gettato via, mentre il programma L0 viene caricato nella memoria del computer per essere eseguito. Durante l'esecuzione il computer ha il controllo del nuovo programma L0.

Nell'interpretazione ciascuna istruzione L1 viene esaminata e decodificata, e quindi eseguita direttamente senza generare alcun programma tradotto. In questo caso il computer ha il controllo del programma interprete, mentre il programma L1 è visto come un insieme di dati. Entrambi i metodi, oltre a una sempre più frequente combinazione dei due, sono ampiamente utilizzati.

Invece di ragionare in termini di traduzione e interpretazione, spesso è più semplice immaginare l'esistenza di un ipotetico computer o **macchina virtuale**, il cui linguaggio macchina sia L1. Chiamiamo questa macchina virtuale M1 (e chiamiamo M0 la macchina virtuale corrispondente al linguaggio L0). Se una tale macchina potesse essere costruita in modo sufficientemente economico, non ci sarebbe affatto bisogno del linguaggio L0 né di una macchina capace di eseguire programmi L0. Gli utenti potrebbero semplicemente scrivere i loro programmi in L1, che verrebbero eseguiti direttamente dal computer. Anche se una macchina virtuale con linguaggio macchina L1 è troppo costosa o complicata da essere costruita realmente, si possono tuttavia scrivere programmi dedicati. Questi programmi possono essere interpretati oppure tradotti da un programma scritto in L0, eseguibile direttamente sui computer esistenti. In altre parole, si possono scrivere programmi per macchine virtuali come se queste esistessero veramente.

Per rendere la traduzione o l'interpretazione utilizzabili in pratica, i linguaggi L0 e L1 non devono essere "troppo" diversi fra loro. Per la maggior parte delle applicazioni questo vincolo fa sì che L1, pur essendo migliore di L0, sia spesso ancora lontano dal linguaggio ideale. Questo risultato può sembrare forse scoraggiante rispetto alle intenzioni iniziali, secondo cui la creazione di L1 avrebbe dovuto sollevare il programmatore dall'onere di esprimere algoritmi in un linguaggio più adatto alle macchine che agli utenti. In ogni caso la situazione non è senza speranza.

L'approccio più ovvio consiste nell'inventare un nuovo insieme d'istruzioni che sia, rispetto a L1, maggiormente orientato agli utenti piuttosto che alle macchine. Anche questo terzo insieme forma a sua volta un linguaggio, che chiamiamo L2 (e la cui corrispondente macchina virtuale sarà M2). Si possono scrivere direttamente programmi in L2 come se esistesse realmente una macchina virtuale dotata di tale linguaggio macchina. Questi programmi possono essere tradotti in L1 oppure eseguiti da un interprete scritto in L1.

La definizione di una serie di linguaggi, ciascuno dei quali più pratico da utilizzare rispetto al precedente, può continuare indefinitamente finché non se ne ottenga uno sufficientemente adeguato. Ciascun linguaggio utilizza il precedente come base; un computer che usa questa tecnica può quindi essere immaginato come una serie di **strati** o **livelli** disposti l'uno sopra l'altro, come mostrato nella Figura 1.1. Il livello, o linguaggio, che si trova più in basso è il più semplice, mentre quello più in alto è il più sofisticato.

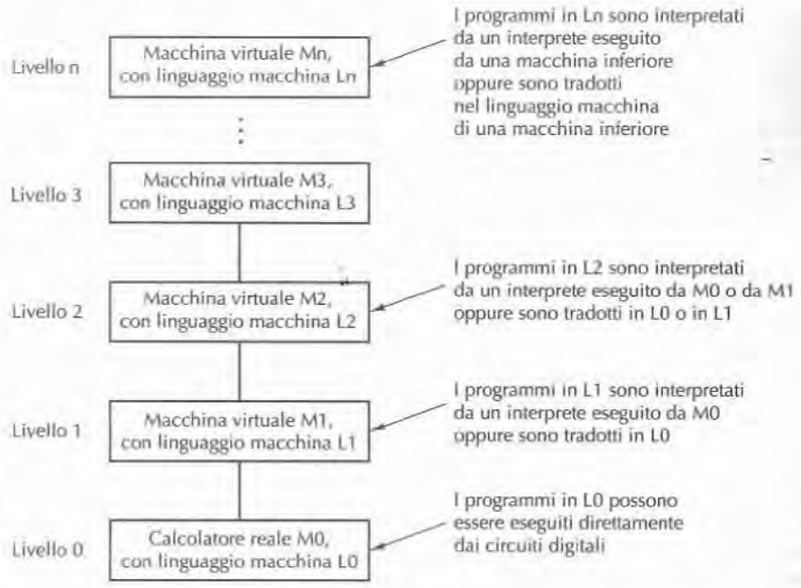


Figura 1.1 Macchina multilivello.

Tra i linguaggi e le macchine virtuali sussiste un'importante relazione. Una macchina ha un proprio linguaggio macchina, costituito da tutte le istruzioni che è in grado di eseguire. Di fatto, una macchina definisce un linguaggio. Viceversa, un linguaggio definisce una macchina, vale a dire quella che è in grado di eseguire tutti i programmi scritti in quel linguaggio. Anche se una macchina definita da un linguaggio arbitrario potrebbe ovviamente essere enormemente complicata e costosa da realizzare, è tuttavia possibile immaginarla. Una macchina dotata di C, C++ o Java come linguaggio macchina risulterebbe sicuramente complessa ma, grazie alla tecnologia odierna, sarebbe in ogni caso realizzabile. Vi è tuttavia una buona ragione per non costruire tale computer: risulterebbe infatti più costoso rispetto all'utilizzo di altre tecniche. Il semplice fatto che sia realizzabile non è sufficiente, dato che un progetto, per essere fattibile, deve anche essere economicamente conveniente.

In un certo senso, un computer con n livelli può essere interpretato come n distinte macchine virtuali; d'ora in avanti utilizzeremo quindi in modo interscambiabile i termini "livello" e "macchina virtuale". Solo i programmi scritti nel linguaggio L0 possono essere eseguiti direttamente dalla componentistica digitale senza far ricorso alla traduzione o all'interpretazione, mentre i programmi scritti in L1, L2, ..., Ln devono essere interpretati da un interprete eseguito a un livello inferiore oppure devono essere tradotti in un altro linguaggio, corrispondente a un livello più basso.

Se si scrivono programmi utilizzando la macchina di un dato livello non ci si deve preoccupare degli interpreti o dei traduttori sottostanti, dato che la struttura della macchina assicura che questi programmi saranno, in un modo o nell'altro, eseguiti. Non è di alcun interesse pratico il fatto che siano portati a termine da un interprete, eseguito a sua volta da un altro interprete, oppure che siano eseguiti direttamente dai circuiti digitali. In entrambi i casi si ottiene lo stesso risultato: i programmi vengono eseguiti.

La maggior parte dei programmatori che usa una macchina di un dato livello è interessata solamente al livello superiore, quello che meno assomiglia al linguaggio macchina del livello più basso. Tuttavia chi è interessato a comprendere il reale funzionamento di un computer li deve studiare tutti. Anche chi progetta nuovi computer o nuovi livelli (cioè nuove macchine virtuali) deve avere la stessa familiarità con tutti i livelli e non solo con quello superiore. Il tema centrale di questo libro è costituito da concetti e tecniche inerenti la costruzione di macchine come gerarchie di livelli, nonché dai dettagli di quest'ultimi.

1.1.2 Attuali macchine multilivello

La maggior parte dei moderni computer consiste di due o più livelli. Come mostrato nella Figura 1.2 esistono anche macchine costituite da sei livelli. Il livello 0, che si trova alla base, rappresenta il vero e proprio hardware della macchina, i cui circuiti eseguono i programmi scritti nel linguaggio macchina del livello 1. Per completezza occorre dire che esiste un ulteriore livello al di sotto di quello che noi consideriamo livello 0. Questo livello è chiamato **livello dei dispositivi** e non è mostrato nella Figura 1.2, dato che ricade nel campo dell'ingegneria elettronica (al di fuori quindi dell'ambito di questo libro). A questo livello i progettisti hanno a che fare con i singoli transistor, gli elementi primitivi nella progettazione dei computer. Se si vuole studiare il funzionamento interno dei transistor si deve entrare nel campo della fisica dello stato solido.

Gli oggetti che analizzeremo nel livello più basso della nostra trattazione, il **livello logico digitale**, sono le **porte** (*gate*). Queste, pur essendo costruite utilizzando componenti ana-

Figura 1.2 Computer a sei livelli. Sotto ciascun livello è indicato il metodo di supporto (oltre al nome del programma corrispondente).

logici come i transistor, possono essere correttamente modellate come dispositivi digitali. Ciascuna porta, costituita al più da una manciata di transistor, è dotata di uno o più input digitali (segnali corrispondenti ai valori 0 o 1) e calcola in output una semplice funzione dei valori d'ingresso, per esempio AND od OR. È possibile combinare un piccolo numero di porte per formare una memoria a 1 bit, che può memorizzare i valori 0 e 1. Combinando le memorie a 1 bit in gruppi, per esempio di 16, 32 o 64, è possibile formare quelli che vengono chiamati registri. Ciascun **registro** può contenere un numero il cui valore può variare fino a un certo limite. È inoltre possibile combinare le porte per formare lo stesso motore computazionale principale. Nel Capitolo 3 esamineremo in dettaglio le porte e il livello logico digitale.

Subito dopo troviamo il **livello di microarchitettura**. Qui vi è una memoria locale, formata da un gruppo di registri (in genere da 8 a 32), e un circuito chiamato **ALU** (*Arithmetic Logic Unit*, unità aritmetico-logica), capace di effettuare semplici operazioni aritmetiche. I registri sono connessi alla ALU per formare un **percorso dati** lungo il quale questi ultimi si spostano. L'operazione base del percorso dati consiste nel selezionare uno o due registri, permettere alla ALU di operare su di loro (per esempio sommandoli) e memorizzare infine il risultato in uno dei registri. In alcune macchine le operazioni del percorso dati sono controllate da un programma chiamato **microprogramma**, mentre su altre il percorso dati è controllato direttamente dall'hardware. Nelle precedenti edizioni di questo libro abbiamo chiamato tale livello il "livello di microprogrammazione", dato che in passato era quasi sempre rappresentato da un interprete software. Attualmente invece il percorso dati è spesso controllato in modo diretto dall'hardware (interamente o parzialmente); per rispecchiare questo cambiamento abbiamo modificato il nome del livello.

Nelle macchine in cui il controllo del percorso dati avviene via software il microprogramma è un interprete per le istruzioni del livello 2, che utilizza il percorso dati per prelevare, esaminare ed eseguire un'istruzione alla volta. Nel caso per esempio di un'istruzione ADD, questa viene prelevata, i suoi operandi localizzati e portati nei registri, la somma calcolata dalla ALU e infine il risultato viene ricopiato nel registro appropriato. Passi analoghi si verificano anche in una macchina con controllo via hardware del percorso dati, senza però avere un programma memorizzato e dedicato all'interpretazione delle istruzioni del livello 2.

Il livello 2 è costituito da quello che chiamiamo il **livello ISA** (*Instruction Set Architecture Level*, livello di architettura dell'insieme d'istruzioni). Ogni produttore di computer pubblica un manuale per ciascuno dei propri modelli, un "Manuale di riferimento del linguaggio macchina" o "Principi di funzionamento del computer Western Wombat Model 100X", o qualcosa di simile. Questi manuali trattano il livello ISA, ma non quelli sottostanti; quando descrivono l'insieme d'istruzioni della macchina, presentano in realtà le istruzioni eseguite in modo interpretato dal microprogramma o dai circuiti elettronici. Se un produttore di computer dotasse una delle proprie macchine di due diversi interpreti, dovrebbe fornire due distinti manuali del "linguaggio macchina" dato che i due interpreti definirebbero due livelli ISA differenti.

Il livello successivo è generalmente un livello ibrido. La maggior parte delle istruzioni nel suo linguaggio fa parte anche del livello ISA (non vi è ragione per cui un'istruzione che appaia a un livello non possa essere presente anche in altri). Inoltre, vi è un insieme di nuove istruzioni, una diversa organizzazione della memoria e la capacità di eseguire programmi in modo concorrente, oltre a varie altre funzionalità. Fra i diversi modi di progettare il livello 3 vi è una maggior varietà rispetto a quanto non si riscontri nei livelli 1 e 2.

I nuovi servizi aggiunti nel livello 3 sono realizzati da un interprete eseguito al livello 2, storicamente chiamato sistema operativo. Le istruzioni del livello 3, identiche a quelle del livello 2, sono eseguite direttamente dal microprogramma (o dai circuiti elettronici) e non dal sistema operativo. In altre parole, alcune delle istruzioni del livello 3 sono interpretate dal sistema operativo, mentre altre sono interpretate in modo diretto dal microprogramma; per questo motivo tale livello viene detto "ibrido". Nel corso del libro ci si riferirà a questo livello come al **livello macchina del sistema operativo**.

Tra i livelli 3 e 4 vi è una spaccatura fondamentale. I tre livelli inferiori non sono progettati per essere utilizzati dal programmatore medio, ma concepiti principalmente per eseguire interpreti e traduttori necessari come supporto per i livelli più alti. Questi interpreti e traduttori sono scritti da professionisti chiamati **programmatori di sistema**, specializzati nella progettazione e nell'implementazione di nuove macchine virtuali. Il livello 4 e quelli superiori sono invece pensati per i programmatori che devono risolvere problemi applicativi.

Un altro cambiamento che si riscontra a partire dal livello 4 è il modo in cui vengono supportati i livelli superiori. Mentre i livelli 2 e 3 sono sempre interpretati, i livelli 4, 5 e quelli superiori sono generalmente supportati mediante traduzione.

Un'ulteriore differenza tra i primi tre livelli e gli altri riguarda la natura del linguaggio corrispondente. I linguaggi macchina dei livelli 1, 2 e 3 sono numerici; i loro programmi consistono quindi in lunghe serie di numeri, certamente più adatte alle macchine che agli esseri umani. A partire dal livello 4 i linguaggi contengono invece parole e abbreviazioni umanamente comprensibili.

Il livello 4, quello del linguaggio assemblativo, è in realtà una forma simbolica di uno dei linguaggi sottostanti. Questo livello fornisce ai programmatori un modo per scrivere programmi per i livelli 1, 2 e 3 in una forma meno difficoltosa rispetto a quella dei linguaggi delle rispettive macchine virtuali. I programmi in linguaggio assemblativo sono inizialmente tradotti nei linguaggi dei livelli 1, 2 o 3 e successivamente interpretati dall'appropriata macchina virtuale o reale. Il programma che esegue la traduzione è chiamato **assemblatore**.

Il livello 5 consiste generalmente di linguaggi, spesso chiamati **linguaggi ad alto livello**, definiti per essere utilizzati dai programmatori di applicazioni. Ne esistono letteralmente a centinaia; alcuni fra i più conosciuti sono C, C++, Java, LISP e Prolog. I programmi scritti in questi linguaggi sono generalmente tradotti al livello 3 o al livello 4 da un traduttore detto **compilatore**; in casi meno frequenti è anche possibile che essi siano interpretati. I programmi in Java, per esempio, di solito sono tradotti inizialmente in un linguaggio di tipo ISA chiamato *Java byte code*, che viene successivamente interpretato.

In alcuni casi, il livello 5 consiste di un interprete specifico per un determinato campo di applicazione, per esempio la matematica simbolica. Fornisce dati e operazioni per risolvere problemi in quell'area in termini facilmente comprensibili agli esperti di quel campo.

Riassumendo, il concetto chiave da ricordare è che i computer sono progettati come una serie di livelli, ciascuno costruito su quelli che lo precedono. Ciascun livello rappresenta una diversa astrazione, caratterizzata dalla presenza di oggetti e operazioni diversi. Progettando e analizzando i computer in questo modo è possibile tralasciare temporaneamente i dettagli irrilevanti, riducendo di conseguenza un soggetto complesso in qualcosa di più facile comprensione.

L'insieme dei tipi di dati, delle operazioni e delle funzionalità di ciascun livello è chiamato **architettura**. Questa tratta gli aspetti visibili agli utenti di quel determinato livello. Fanno quindi parte dell'architettura le caratteristiche che un programmatore può vedere, come la quantità di memoria disponibile, mentre non ne fanno parte gli aspetti implementativi, come la tecnologia con cui è realizzata la memoria. Lo studio di come progettare le parti di un computer visibili ai programmatori è chiamato **architettura degli elaboratori**. Nella pratica comune architettura dei calcolatori e organizzazione dei computer significano essenzialmente la stessa cosa.

1.1.3 Evoluzione delle macchine multilivello

Per fornire una panoramica delle macchine multilivello, ne esamineremo brevemente gli sviluppi storici, mostrando come il numero e la natura dei livelli siano evoluti nel corso degli anni. I programmi scritti in un vero linguaggio macchina (livello 1) possono essere eseguiti direttamente dai circuiti del computer (livello 0), senza far ricorso ad alcuna traduzione o interpretazione. Questi circuiti formano, insieme alla memoria e ai dispositivi di input/output, l'**hardware** del computer. L'hardware consiste di oggetti tangibili (circuiti integrati, schede con circuiti stampati, cavi, trasformatori, memorie e stampanti) piuttosto che di idee astratte, algoritmi o istruzioni.

Al contrario, il **software** consiste di **algoritmi** (istruzioni dettagliate su come realizzare un determinato compito) e delle loro rappresentazioni per i computer, vale a dire i programmi. I programmi possono essere memorizzati su hard disk, floppy disk, CD-ROM o altri supporti, ma l'essenza del software è l'insieme delle istruzioni che costituiscono il programma, non il supporto fisico su cui sono registrate.

Nei primissimi computer, il confine tra hardware e software era chiaro e cristallino. Nel corso del tempo, man mano che i computer si sono evoluti, quel confine si è però sfocato in modo considerevole per via dell'aggiunta, della rimozione e dell'unione di livelli, tanto che oggi è spesso difficile tenerli separati (Vahid, 2003). E proprio per questo, un tema centrale di questo libro è:

hardware e software sono logicamente equivalenti.

Ogni operazione eseguita via software può anche essere costruita direttamente in hardware, preferibilmente dopo averne compreso in modo sufficientemente approfondito il funzionamento. Come ha affermato Karen Panetta Lentz: "L'hardware non è che software pietrificato". Ovviamente è anche vero il contrario: ogni istruzione eseguita dall'hardware può essere simulata via software. La scelta di collocare certe funzioni nell'hardware e altre nel software è basata su fattori quali il costo, la velocità, l'affidabilità e la frequenza di modifiche che ci si aspetta. Esistono poche regole ferree nello stabilire se X deve andare nell'hardware e Y deve essere programmato esplicitamente. Queste scelte cambiano a seconda del mercato delle tecnologie, della domanda e del modo di usare il computer.

Invenzione della microprogrammazione

I primi computer digitali, risalenti agli anni '40, avevano solamente due livelli: il livello ISA, in cui erano realizzati tutti i programmi, e il livello logico digitale, che li eseguiva. I circuiti del livello logico digitale erano complessi, difficili da comprendere e da realizzare e, oltretutto, inaffidabili.

Nel 1951 Maurice Wilkes, un ricercatore della Cambridge University, propose di progettare un computer a tre livelli, al fine di semplificare drasticamente l'hardware (Wilkes, 1951). Questa macchina avrebbe dovuto avere un interprete predefinito e non modificabile (il microprogramma), la cui funzione sarebbe stata quella di eseguire programmi al livello ISA mediante interpretazione. Sarebbe stato necessario un minor numero di circuiti elettronici, dato che l'hardware avrebbe dovuto eseguire microprogrammi costituiti da un insieme limitato di istruzioni invece di programmi a livello ISA, basati su un insieme di istruzioni molto più ampio. Dato che i circuiti di allora erano costituiti da valvole[1], una tale semplificazione prometteva di ridurne il numero migliorando di conseguenza l'affidabilità (cioè diminuendo il numero giornaliero di guasti).

Durante gli anni '50 vennero costruite poche di queste macchine a tre livelli, mentre lo furono di più negli anni '60. A partire dal 1970 l'idea di avere un livello ISA interpretato da un microprogramma divenne dominante e tutte le principali macchine dell'epoca la implementarono.

Invenzione del sistema operativo

In questi primi anni la maggior parte dei computer era di tipo "fai da te", il che significava che il programmatore doveva far funzionare la macchina personalmente. Vicino a ogni macchina vi era un foglio su cui ogni programmatore prenotava la fascia oraria durante la quale desiderava far eseguire il proprio programma, per esempio dalle 3 alle 5 del mattino (molti

[1] I cosiddetti tubi a vuoto (*N.d.Rev.*).

programmatori preferivano lavorare quando vi era silenzio in sala macchine). Quando arrivava il momento, il programmatore si dirigeva verso la sala macchine con un mazzo di schede perforate da 80 colonne (uno dei primi supporti di input) in una mano e una matita appuntita nell'altra. Una volta arrivato nella stanza del computer spingeva gentilmente il precedente programmatore fuori dalla porta e prendeva il controllo del computer.

Se il programmatore desiderava eseguire un programma FORTRAN si verificavano i seguenti passi.

1. Guardava nell'armadietto dove erano tenuti tutti i programmi, estraeva il grande mazzo di schede verdi etichettato "compilatore FORTRAN", lo inseriva nel lettore delle schede perforate e premeva il pulsante START.
2. Inseriva il suo programma nel lettore delle schede e premeva il pulsante CONTINUE. Il programma veniva letto.
3. Quando il computer si arrestava, il programmatore faceva leggere il suo programma FORTRAN una seconda volta. Anche se alcuni compilatori richiedevano una sola passata sull'input, molti altri ne richiedevano due o più. A ogni passata doveva essere letto un gran numero di schede.
4. Infine la traduzione si avvicinava al termine. Spesso in questa fase il programmatore diventava nervoso, poiché, nel caso in cui il compilatore avesse trovato un errore nel programma, avrebbe dovuto ricominciare da capo l'intero processo. Se non vi erano errori, il compilatore perforava una scheda contenente il programma tradotto in linguaggio macchina.
5. Il programmatore inseriva quindi il programma in linguaggio macchina e il gruppo di schede delle subroutine nel lettore delle schede e le faceva leggere tutte insieme.
6. Il programma iniziava l'esecuzione; spesso non funzionava o si bloccava a metà. In genere il programmatore si metteva a manipolare gli interruttori e fissava per un attimo le luci della console. Se era fortunato riusciva a capire il problema, correggeva l'errore e tornava all'armadietto contenente il compilatore FORTRAN per ricominciare la procedura da capo. Se era meno fortunato allora doveva stampare una mappa della memoria, chiamata **dump di memoria** (*core dump*) e portarsela a casa da analizzare.

Questa procedura, al di là di piccole variazioni, fu per anni la normalità in molti centri di calcolo. I programmatori erano costretti a imparare come far funzionare la macchina e a sapere che cosa fare quando si verificava un guasto (cosa che succedeva spesso). La macchina restava frequentemente inattiva mentre gli operatori portavano le schede in giro per la stanza o si grattavano la testa nel tentativo di comprendere il motivo per cui i loro programmi non funzionavano correttamente.

Intorno al 1960 si cercò di ridurre la quantità di tempo sprecato, automatizzando il lavoro dell'operatore. Un programma chiamato **sistema operativo** era tenuto costantemente nel computer. Il programmatore forniva, insieme al proprio programma, alcune schede di controllo che venivano lette ed eseguite dal sistema operativo. La Figura 1.3 mostra un semplice compito (*job*) per uno dei primi e più diffusi sistemi operativi, chiamato FMS (*Fortran Monitor System*) e funzionante su un IBM 709.

Figura 1.3 Semplice "job" per il sistema operativo FMS.

Il sistema operativo leggeva la scheda *JOB e usava le informazioni ivi memorizzate per la gestione degli *account* (l'asterisco era utilizzato per identificare le schede di controllo, in modo da non confonderle con quelle del programma e dei dati); successivamente leggeva la scheda *FORTRAN, che era un'istruzione per caricare il compilatore FORTRAN da un nastro magnetico. Il compilatore poteva quindi leggere e compilare il programma FORTRAN. Quando il compilatore terminava, restituiva il controllo al sistema operativo, che iniziava la lettura della scheda *DATA. Questa rappresentava un'istruzione per eseguire il programma tradotto, utilizzando come dati le schede che seguivano la scheda *DATA.

Sebbene il sistema operativo fosse stato progettato per automatizzare il compito dell'operatore (da qui il suo nome), fu allo stesso tempo il primo passo nello sviluppo di una nuova macchina virtuale. La scheda *FORTRAN potrebbe essere interpretata come un'istruzione virtuale "compila il programma". Analogamente la scheda *DATA potrebbe essere vista come un'istruzione virtuale "esegui il programma". Anche se due sole istruzioni non sono sufficienti per costituire un vero e proprio livello, era comunque il primo passo in quella direzione.

Negli anni successivi, i sistemi operativi divennero sempre più sofisticati. Al livello ISA vennero aggiunte nuove istruzioni, nuovi servizi e nuove caratteristiche, finché non cominciò ad assumere l'aspetto di un nuovo livello. Alcune di queste istruzioni erano identiche a quelle del livello ISA, ma altre, in particolare le istruzioni di input/output, erano completamente diverse. Le nuove istruzioni erano spesso conosciute come **macro del sistema operativo** o **chiamate al supervisore**; oggi si usa generalmente il termine **chiamate di sistema**.

I sistemi operativi si svilupparono anche sotto altri aspetti. I primi leggevano i pacchi di schede e stampavano l'output attraverso la stampante; questa organizzazione era conosciuta come **sistema batch** ("a lotti"). Generalmente vi era un'attesa di varie ore tra il momento in cui un programma veniva assegnato alla macchina e quello in cui il risultato era pronto; in tali circostanze lo sviluppo del software risultava difficile.

Nei primi anni '60 alcuni ricercatori del Dartmouth College, del M.I.T. e di altre università svilupparono sistemi operativi che permettevano a più programmatori di comunicare direttamente con il computer. In questi sistemi una serie di terminali remoti erano connessi attraverso linee telefoniche al computer centrale, che veniva così condiviso da diversi utenti. Un programmatore poteva digitare un programma e ottenere il risultato quasi immediatamente, nel proprio ufficio, nel garage di casa o ovunque si trovasse il terminale. Questi sistemi erano chiamati **sistemi a condivisione di tempo** (*time sharing systems*).

Più che agli aspetti riguardanti la condivisone di tempo, il nostro interesse per i sistemi operativi è rivolto a quelle parti che interpretano le istruzioni e le funzionalità presenti al livello 3, ma non al livello ISA. Occorre tenere a mente che i compiti dei sistemi operativi si spingono molto al di là della semplice interpretazione delle funzionalità aggiunte al livello ISA.

Migrazione delle funzionalità verso il microcodice

Quando la microprogrammazione divenne una pratica comune (a partire dal 1970), i progettisti capirono che avrebbero potuto aggiungere nuove istruzioni semplicemente estendendo il microprogramma. In altre parole avrebbero potuto aggiungere "hardware" (nuove istruzioni macchina) per mezzo della programmazione. Questa scoperta portò a una virtuale esplosione dell'insieme delle istruzioni macchina, dato che i progettisti gareggiavano nel produrre insiemi d'istruzioni sempre più estesi e migliori. Molte di queste istruzioni non erano strettamente necessarie nel senso che il loro risultato poteva essere facilmente ottenuto attraverso una sequenza d'istruzioni già esistenti, ma spesso la loro esecuzione risultava leggermente più veloce. Molte macchine, per esempio, avevano un'istruzione `INC` (INCrement) che sommava uno a un numero; una tale istruzione speciale per aggiungere sempre 1 (o, analogamente, 720) non era realmente necessaria, dato che queste macchine erano dotate di una generale istruzione di addizione, `ADD`. Tuttavia l'istruzione `INC` era in genere leggermente più veloce di `ADD`, e per questo venne introdotta.

Per la stessa ragione furono aggiunte molte altre istruzioni, tra cui era spesso possibile trovare le seguenti:

1. istruzioni per la moltiplicazione e la divisione di interi
2. istruzioni aritmetiche in virgola mobile
3. istruzioni per la chiamata e il ritorno da procedure
4. istruzioni per velocizzare i cicli
5. istruzioni per trattare stringhe di caratteri.

Inoltre, quando i progettisti videro quanto era facile aggiungere nuove istruzioni, cominciarono a guardarsi intorno per trovare altre caratteristiche da inserire nei loro microprogrammi. Alcuni esempi di queste nuove aggiunte furono:

1. funzionalità per accelerare calcoli sugli array (indicizzazione e indirizzamento indiretto)
2. funzionalità per permettere ai programmi di essere spostati in memoria dopo l'inizio della loro esecuzione (servizi di rilocazione)

3. sistemi di interrupt che segnalano al computer che un'operazione di input o di output è completata
4. possibilità di sospendere un programma e farne partire un altro attraverso poche istruzioni (commutazione di processo)
5. istruzioni speciali per l'elaborazione di audio, immagini e file multimediali.

Allo stesso modo nel corso degli anni vennero aggiunte numerose altre caratteristiche e funzionalità, generalmente con lo scopo di velocizzare alcune attività particolari.

Eliminazione della microprogrammazione

Durante gli anni d'oro della microprogrammazione (gli anni '60 e '70) le dimensioni dei microprogrammi aumentarono così tanto che cominciarono a divenire via via più lenti. A un certo punto alcuni ricercatori compresero che le macchine potevano essere accelerate eliminando il microprogramma, riducendo considerevolmente l'insieme delle istruzioni e facendo in modo che quelle rimanenti venissero eseguite direttamente (cioè attraverso un controllo hardware del percorso dati). Sotto un certo punto di vista la progettazione di computer ha compiuto un cerchio completo, ritornando alla situazione precedente al momento in cui Wilkes introdusse la microprogrammazione.

La ruota continua però a girare. I programmi Java sono generalmente eseguiti compilandoli in un linguaggio intermedio (*Java byte code*) e poi interpretando il *Java byte code*.

La nostra analisi mira a mostrare che il confine tra hardware e software è arbitrario e cambia continuamente; il software di oggi potrebbe essere l'hardware di domani e viceversa. Per di più anche la separazione tra i vari livelli è fluida. Dal punto di vista del programmatore non è importante sapere come un'istruzione sia effettivamente implementata (eccetto forse per la sua velocità di esecuzione). Chi programma al livello ISA può usare la sua istruzione di moltiplicazione come se fosse un'istruzione hardware, senza preoccuparsene o senza neanche essere a conoscenza se lo sia realmente oppure no. L'hardware è concepibile come tale da alcuni, come software da altri. Nel seguito del libro ritorneremo su questi temi.

1.2 Pietre miliari nell'architettura dei computer

Nel corso dell'evoluzione che ha portato agli attuali computer, sono stati progettati e costruiti centinaia di diversi tipi di elaboratori. La maggior parte è stata dimenticata da tempo, mentre un ristretto numero ha avuto un impatto significativo sulle idee alla base delle moderne architetture. In questo paragrafo faremo un rapido excursus su alcuni degli sviluppi storici più significativi, al fine di poter comprendere al meglio come si è arrivati allo stato attuale. È ovvio che qui tratteremo soltanto i punti principali di questo percorso, tralasciandone al contempo molti altri. La Figura 1.4 elenca alcune delle macchine che rappresentano delle pietre miliari e che saranno trattate in questo paragrafo. Slater (1987) è un ottimo riferimento per cercare ulteriore materiale storico sui fondatori dell'era dei calcolatori; si consulti invece il libro illustrato di Morgan (1997) se si cercano sintetiche biografie e belle foto a colori (realizzate da Louis Fabian Bachrach) dei protagonisti che hanno dato inizio all'era dei computer.

Anno	Nome	Realizzato da	Commento
1834	Analytical Engine	Babbage	Primo tentativo di costruire un computer digitale
1936	Z1	Zuse	Prima macchina calcolatrice funzionante basata su relè
1943	COLOSSUS	Governo britannico	Primo computer elettronico
1944	Mark I	Aiken	Primo computer americano di uso generale
1946	ENIAC I	Eckert/Mauchley	Inizio della moderna storia dei computer
1949	EDSAC	Wilkes	Primo computer a programma memorizzato
1951	Whirlwind I	M.I.T.	Primo computer con elaborazione in tempo reale
1952	IAS	Von Neumann	Progetto su cui si basa la maggior parte delle macchine odierne
1960	PDP-1	DEC	Primo minicomputer (venduto in 50 unità)
1961	1401	IBM	Piccola macchina aziendale di grande diffusione
1962	7094	IBM	Dominò il campo della computazione scientifica nei primi anni '60
1963	B5000	Burroughs	Prima macchina progettata per un linguaggio ad alto livello
1964	360	IBM	Prima linea di prodotti progettata come una famiglia
1964	6600	CDC	Primo supercomputer per calcoli scientifici
1965	PDP-8	DEC	Primo minicomputer con un mercato di massa (venduto in 50.000 unità)
1970	PDP-11	DEC	Dominò il mondo dei minicomputer negli anni '70
1974	8080	Intel	Primo computer a 8 bit di uso generale su un chip
1974	CRAY-1	Cray	Primo supercomputer vettoriale
1978	VAX	DEC	Primo superminicomputer a 32 bit
1981	IBM PC	IBM	Inizio dell'era moderna dei personal computer
1981	Osborne-1	Osborne	Primo computer portatile
1983	Lisa	Apple	Primo personal computer dotato di una GUI
1985	386	Intel	Progenitore a 32 bit della linea Pentium
1985	MIPS	MIPS	Prima macchina RISC commerciale
1987	SPARC	Sun	Prima workstation RISC basata su SPARC
1990	RS6000	IBM	Prima macchina superscalare
1992	Alpha	DEC	Primo personal computer a 64 bit
1993	Newton	Apple	Primo computer palmare

Figura 1.4 Alcune pietre miliari nello sviluppo del computer digitale moderno.

1.2.1 Generazione zero – Computer meccanici (1642-1945)

La prima persona che costruì una macchina calcolatrice funzionante fu lo scienziato francese Blaise Pascal (1623-1662), in onore del quale è stato chiamato l'omonimo linguaggio di programmazione. Pascal progettò e costruì questo dispositivo nel 1642, a 19 anni, per aiutare suo padre, un esattore delle tasse del governo francese. La macchina era interamente meccanica, costituita da ingranaggi e veniva azionata a mano con una manovella.

La macchina di Pascal era in grado di compiere solamente somme e sottrazioni, ma trent'anni dopo un importante matematico tedesco, il Barone Gottfried Wilhelm von Leibniz (1646-1716), costruì un'altra macchina meccanica capace di eseguire anche moltiplicazioni e divisioni. Di fatto Leibniz costruì, già tre secoli fa, l'equivalente di una piccola calcolatrice tascabile.

Per 150 anni non successe niente di significativo finché un professore di matematica della Cambridge University, Charles Babbage (1792-1871), inventore tra l'altro del tachimetro, progettò e costruì una macchina chiamata *difference engine* ("macchina differenziale"). Questo dispositivo meccanico, capace come quello di Pascal soltanto di sommare e sottrarre, fu progettato per calcolare tabelle di numeri utili per la navigazione. L'intera costruzione della macchina fu pensata per eseguire un solo algoritmo, il metodo matematico delle differenze finite. L'aspetto più interessante della *difference engine* fu però la sua forma di output: i risultati venivano incisi mediante una punta d'acciaio su una lastra di rame, anticipando quindi i futuri supporti a singola scrittura quali le schede perforate e i CD-ROM.

Sebbene la *difference engine* funzionasse in modo abbastanza soddisfacente, Babbage si stancò presto di una macchina capace di eseguire un solo algoritmo. Egli cominciò a profondere una sempre maggior quantità di tempo e di beni familiari (per non parlare delle 17.000 sterline di fondi governativi) nella progettazione e costruzione di una macchina che ne rappresentasse l'evoluzione: la *analytical engine* ("macchina analitica"). La *analytical engine* era composta da quattro componenti: il magazzino (la memoria), il mulino (l'unità computazionale), il dispositivo di input (le schede perforate) e il dispositivo di output (output stampato e perforato). Il magazzino era composto da 1000 parole da 50 cifre, utilizzate per memorizzare dati e risultati. Il mulino poteva prelevare gli operandi dal magazzino per sommarli, sottrarli, moltiplicarli o dividerli e infine memorizzarne nuovamente il risultato nel magazzino. Anche questa macchina, al pari della *difference engine*, era interamente meccanica.

Il grande avanzamento rappresentato dall'*analytical engine* consisteva nel fatto di essere di uso generale. Leggeva le istruzioni dalle schede perforate e le eseguiva. Alcune istruzioni comandavano la macchina in modo che prelevasse due numeri dal magazzino, li portasse nel mulino, eseguisse su di loro una data operazione (per esempio la somma) e ne rispedisse il risultato al magazzino. Altre istruzioni erano in grado di analizzare un numero e ramificare l'esecuzione del programma in base al suo segno. Perforando un diverso programma sulle schede di input era quindi possibile far eseguire all'*analytical engine* diverse computazioni, cosa che era invece impossibile con la *difference engine*.

Dato che la *analytical engine* era programmabile mediante un semplice linguaggio assemblativo, era necessario produrre il software. A tal fine, Babbage assunse Ada Augusta Lovelace, la giovane figlia del famoso poeta inglese, Lord Byron. Ada Lovelace fu quindi la prima programmatrice della storia e in suo onore fu chiamato il linguaggio di programmazione Ada.

Sfortunatamente, come avviene nel caso di molti progettisti moderni, Babbage non riuscì mai a ottenere un hardware completamente privo di errori. Il problema nasceva dalla necessità di avere migliaia e migliaia di rotelle e ingranaggi per ottenere un grado di precisione che la tecnologia del diciannovesimo secolo non era ancora in grado di fornire. Ciononostante le sue idee erano in largo anticipo sui tempi, al punto che la maggior parte dei computer moderni presenta oggi una struttura molto simile all'*analytical engine*. Per questo motivo è giusto affermare che Babbage fu il padre (o meglio, il nonno) degli odierni calcolatori.

Il successivo e significativo avanzamento avvenne verso la fine degli anni '30, quando uno studente tedesco di ingegneria, Konrad Zuse, costruì una serie di macchine calcolatrici automatiche utilizzando dei relé elettromagnetici. Una volta scoppiata la guerra, non riuscì a ottenere finanziamenti dal governo, poiché i burocrati si aspettavano di vincere il conflitto in così breve tempo da ritenere che la nuova macchina non sarebbe stata pronta prima della fine della guerra. Zuse non conosceva il lavoro di Babbage e le sue macchine furono distrutte nel 1944 durante un bombardamento degli Alleati su Berlino. Anche se per questi motivi il suo lavoro non poté influenzare lo sviluppo delle macchine successive, Zuse va comunque considerato come uno dei pionieri in questo campo.

Non molto tempo dopo, negli Stati Uniti altri due ricercatori, John Atanasoff dell'Iowa State College e George Stibbitz dei Bell Labs, si dedicarono alla progettazione di calcolatori. La macchina di Atanasoff era incredibilmente avanzata per l'epoca: era basata sull'aritmetica binaria e utilizzava dei condensatori per la memoria. Affinché la loro carica non si disperdesse, la memoria veniva periodicamente aggiornata, secondo un processo che Atanasoff chiamò *jogging the memory*. Le moderne memorie dinamiche (DRAM) funzionano secondo lo stesso principio. Sfortunatamente la macchina non divenne mai realmente operativa; in un certo senso Atanasoff fu come Babbage, cioè un precursore che non ebbe successo a causa dell'inadeguatezza tecnologica del proprio tempo.

Al contrario il computer di Stibbitz, sebbene meno evoluto di quello di Atanasoff, riuscì a funzionare e il suo inventore ne diede una dimostrazione pubblica nel 1940 durante una conferenza al Dartmouth College. Tra il pubblico c'era John Mauchley, a quel tempo ancora sconosciuto, ma di cui in seguito si sarebbe sentito parlare nel mondo dei computer. Mentre Zuse, Stibbitz e Atanasoff stavano progettando calcolatori automatici, un giovane di nome Howard Aiken si sforzava di svolgere a mano tediosi calcoli numerici, per la sua tesi di dottorato, alla Harvard University. Dopo la laurea, Aiken comprese l'importanza di poter svolgere calcoli per mezzo di una macchina. Frequentando una biblioteca, scoprì il lavoro di Babbage e decise di costruire per mezzo di relé il computer di uso generale che Babbage a suo tempo non riuscì a realizzare mediante ingranaggi.

Nel 1944, alla Harvard University, Aiken completò la sua prima macchina, il Mark I. Era composta da 72 parole di 23 cifre decimali, aveva un tempo d'istruzione di 6 secondi e usava un nastro di carta perforato per l'input e l'output. Dal momento in cui Aiken realizzò il suo successore, il Mark II, i computer basati su relé divennero obsoleti; era iniziata infatti l'era dell'elettronica.

1.2.2 Prima generazione – Valvole (1945-1955)

Lo stimolo per lo sviluppo dei computer elettronici venne dalla Seconda Guerra Mondiale. Nella prima parte della guerra i sottomarini tedeschi stavano decimando la flotta britannica. Da Berlino gli ammiragli tedeschi spedivano i comandi via radio ai sottomarini; questi ordini potevano quindi essere intercettati dai britannici. Il problema era che i messaggi venivano codificati per mezzo di un dispositivo chiamato **ENIGMA**; il precursore di questa macchina era stato progettato da Thomas Jefferson che, oltre a essere stato il terzo presidente degli Stati Uniti d'America, fu anche un inventore dilettante.

All'inizio della guerra lo spionaggio britannico riuscì a procurarsi una macchina ENIGMA grazie all'aiuto dei servizi segreti polacchi che erano riusciti a rubarla ai tedeschi. Tuttavia, per poter decifrare un messaggio codificato occorreva svolgere un'enorme quantità

di calcoli, ed era necessario poterlo fare molto velocemente, non appena il comando veniva intercettato, affinché ciò potesse essere di una qualche utilità. Per decodificare questi messaggi il governo britannico creò un laboratorio segretissimo per la costruzione di un computer chiamato COLOSSUS. Il famoso matematico inglese Alan Turing diede il suo aiuto alla progettazione di questa macchina. COLOSSUS divenne operativo nel 1943 ma, dato che il governo britannico tenne sotto segreto militare per 30 anni praticamente ogni aspetto del progetto, lo sviluppo di COLOSSUS non ebbe alcun seguito. Vale la pena ricordarlo semplicemente perché fu il primo elaboratore digitale al mondo.

Oltre a distruggere le macchine di Zuse e incentivare la costruzione di COLOSSUS, la guerra condizionò il campo dei computer anche negli Stati Uniti. L'esercito aveva bisogno di tabelle per impostare la mira dell'artiglieria pesante e a tal scopo vennero assunte centinaia di donne (in quanto ritenute più precise degli uomini) affinché producessero in serie queste tabelle per mezzo di calcolatori manuali. Tuttavia la procedura richiedeva molto tempo, e vi era sempre il rischio dell'errore umano.

John Mauchley, che aveva studiato sia i lavori di Atanasoff sia quelli di Stibbitz, venne a sapere che l'esercito era interessato alla costruzione di calcolatori meccanici. Come molti altri ricercatori dopo di lui, avanzò all'esercitò una richiesta di sovvenzione per finanziare la costruzione di un calcolatore elettronico. Nel 1943 la richiesta fu accettata e Mauchley cominciò a costruire, insieme al suo studente J. Presper Eckert, un computer elettronico che chiamarono **ENIAC** (*Electronic Numerical Integrator And Computer*). L'elaboratore ENIAC era costituito da 18.000 valvole termoioniche e 1500 relé, pesava 30 tonnellate e consumava 140 KW di energia. Dal punto di vista dell'architettura, la macchina era dotata di 20 registri, ciascuno dei quali in grado di memorizzare un numero decimale a 10 cifre. Un registro decimale è una memoria molto piccola che può mantenere un valore fino a un massimo numero di cifre digitali, analogamente al contachilometri che tiene traccia di quanta strada una macchina ha percorso nella sua vita. ENIAC veniva programmato regolando 6000 interruttori multi-posizione e connettendo una moltitudine di prese con una vera e propria foresta di cavi.

La macchina fu terminata solo nel 1946, quando ormai non poteva più essere utile per lo scopo originario. Tuttavia, dato che la guerra era finita, fu concesso a Mauchley e Eckert di organizzare una scuola estiva per presentare il loro lavoro ai colleghi. Questo evento segnò l'inizio di un'esplosione di interesse nella costruzione di grandi computer digitali.

Dopo quella scuola estiva molti altri ricercatori intrapresero la costruzione di computer elettronici. Il primo a essere operativo fu l'elaboratore EDSAC (1949), costruito presso la Cambridge University da Maurice Wilkes. Fra gli altri progetti, vi furono JOHNIAC alla Rand Corporation, ILLIAC alla Illinois University, MANIAC presso il Los Alamos Laboratory e WEIZAC al Weizmann Institute in Israele.

Eckert e Mauchley cominciarono presto a lavorare su un successore della loro macchina, chiamato **EDVAC** (*Electronic Discrete Variable Automatic Computer*). Quel progetto morì però nel momento in cui decisero di lasciare la Pennsylvania University per formare a Philadelphia (la Silicon Valley non era ancora stata inventata) una propria società, la Eckert-Mauchley Computer Corporation. Dopo una serie di fusioni questa compagnia è diventata la moderna Unisys Corporation.

Facendo una piccola digressione in ambito legale occorre ricordare che Eckert e Mauchley registrarono un brevetto in cui sostenevano di essere gli inventori del computer digitale. A posteriori si può affermare che non sarebbe stato male possedere un tale brevetto. Dopo anni di controversie legali i tribunali decisero che il brevetto Eckert-Mauchley non era valido e che l'inventore del computer digitale, pur non avendolo mai brevettato, era stato John Atanasoff.

Mentre Eckert e Mauchley erano impegnati a lavorare su EDVAC, un membro del progetto ENIAC, John von Neumann, andò all'Institute of Advanced Studies di Princeton per costruire la propria versione dell'EDVAC, la macchina **IAS**. Von Neumann era un genio del livello di Leonardo da Vinci. Parlava molte lingue, era un esperto di fisica e matematica e aveva la capacità di ricordare perfettamente ogni cosa avesse sentito, visto o letto; era inoltre in grado di citare a memoria e alla lettera libri che aveva letto anni prima. Nel momento in cui cominciò a interessarsi ai computer era già il più importante matematico al mondo.

Una delle cose che gli apparvero subito evidenti fu che la programmazione dei computer mediante un gran numero di interruttori e cavi era lenta, tediosa e poco flessibile. Egli comprese che anche i programmi, al pari dei dati, potevano essere rappresentati in forma numerica all'interno della memoria del computer. Inoltre si rese conto che la contorta aritmetica decimale seriale usata nell'elaboratore ENIAC, in cui ogni cifra era rappresentata da 10 valvole (1 accesa e 9 spente), poteva essere sostituita da un'aritmetica binaria parallela, in modo simile a quanto aveva realizzato Atanasoff anni prima.

Il primo progetto elementare che descrisse è conosciuto al giorno d'oggi come **macchina di von Neumann**. Fu utilizzato nel computer EDSAC, il primo che memorizzava il programma in memoria, ed è ancora oggi, a ben mezzo secolo di distanza, alla base di quasi tutti i computer digitali. Questo progetto, così come la macchina IAS costruita in collaborazione con Herman Goldstine, ha avuto una tale influenza che vale la pena descriverlo brevemente. Nonostante si associ sempre il nome di Von Neumann a questo progetto, anche Goldstine e altri ricercatori diedero un contributo sostanziale. Uno schema dell'architettura è mostrato nella Figura 1.5.

La macchina di von Neumann era composta da cinque componenti principali: la memoria, l'unità aritmetico-logica, l'unità di controllo e i dispositivi di input e output. La memoria era costituita da 4096 locazioni, ciascuna delle quali conteneva 40 bit. Ciascuna pa-

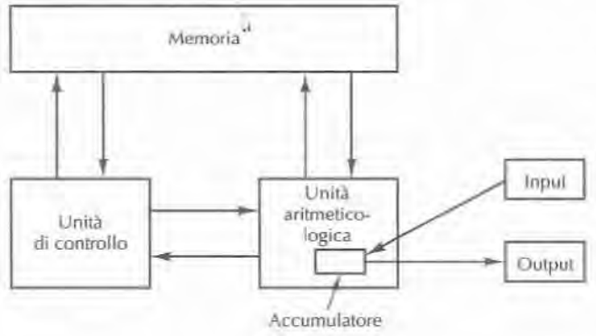


Figura 1.5 La macchina originale di von Neumann.

rola poteva mantenere due istruzioni da 20 bit oppure un numero da 40 bit. Le istruzioni erano composte da 8 bit, che definivano il tipo d'istruzione, e dai restanti 12 bit, che specificavano una delle 4096 parole di memoria. L'unità aritmetico-logica e l'unità di controllo formavano insieme il "cervello" del computer; negli elaboratori moderni esse sono combinate in un unico chip chiamato **CPU** (*Central Processing Unit*, "unità centrale di calcolo").

All'interno dell'unità aritmetico-logica vi era uno speciale registro da 40 bit, chiamato **accumulatore**. La tipica istruzione sommava una parola di memoria all'accumulatore oppure ne copiava il contenuto in memoria. La macchina non aveva un'aritmetica in virgola mobile dato che von Neumann riteneva che ogni matematico che si rispetti dovesse essere in grado di tener traccia a mente della posizione della virgola (per essere precisi, della virgola binaria).

Praticamente nello stesso periodo in cui von Neumann realizzava la macchina IAS, altri ricercatori del M.I.T. stavano costruendo un computer. Diversamente da IAS, ENIAC e altre macchine dello stesso tipo, dotate di lunghe parole e pensate per poter elaborare numeri molto grandi, la macchina del M.I.T., chiamata Whirlwind I, aveva parole di soli 16 bit ed era progettata per il controllo in tempo reale. Grazie a questo progetto si deve l'invenzione della memoria a nuclei magnetici da parte di Jay Forrester e, infine, la nascita del primo minicomputer commerciale.

Mentre avveniva tutto questo, IBM era una piccola società impegnata nel commercio di schede perforate e di macchine per il loro ordinamento. Inizialmente IBM non era fortemente interessata ai computer, sebbene avesse in parte finanziato il lavoro di Aiken. Cominciò a diventarlo nel 1953 dopo la produzione del modello 701 e quindi molto tempo dopo che la compagnia di Eckert e Mauchley era diventata, grazie al computer UNIVAC, il numero uno sul mercato. Il modello 701 era dotato di 2048 parole di 36 bit, con due istruzioni per parola, e fu la prima di una serie di macchine scientifiche che in una decade cominciarono a dominare il mercato. Tre anni dopo fu il momento del modello 704, che inizialmente aveva 4096 parole di memoria centrale, istruzioni a 36 bit e un nuovo, innovativo, hardware in virgola mobile. Nel 1958 IBM cominciò la produzione del modello 709, la sua ultima macchina che utilizzava le valvole, in pratica una miglioria del computer 704.

1.2.3 Seconda generazione – Transistor (1955-1965)

Il transistor è stato inventato nel 1948 presso i Bell Labs da John Bardeen, Walter Brattain e William Shockley, per la cui scoperta ricevettero nel 1956 il premio Nobel per la fisica. In 10 anni il transistor rivoluzionò i computer al punto che nei tardi anni '50 i computer a valvole divennero obsoleti. Il primo computer a transistor fu costruito presso i Lincoln Laboratory del M.I.T. Si trattava di una macchina a 16 bit che seguiva la linea del Whirlwind I. Fu chiamato **TX-0** (*Transistorized eXperimental computer 0*), ideato semplicemente come dispositivo per testare il più evoluto TX-2.

Il computer TX-2 non prese mai il volo, ma nel 1957 Kenneth Olsen, uno degli ingegneri che lavoravano al Lincoln Laboratory, fondò una società, la Digital Equipment Corporation (DEC), per produrre una macchina commerciale molto simile al modello TX-0. Passarono ben quattro anni prima di veder apparire questa macchina, chiamata PDP-1; ciò fu dovuto principalmente al fatto che i finanziatori che avevano fondato la DEC erano fermamente convinti che non vi fosse mercato per i computer. Dopo tutto anche T.J. Watson, ex presidente di IBM, una volta ebbe a dire che il mercato mondiale di computer era all'incirca di sole quattro o cinque unità. Al contrario DEC si affermò principalmente per la vendita di schede di circuiti di piccole dimensioni.

Nel 1961 quando finalmente apparve, l'elaboratore PDP-1 era dotato di 4096 parole da 18 bit e poteva eseguire 200.000 istruzioni al secondo. Queste prestazioni corrispondevano alla metà di quelle dell'IBM 7090, successore a transistor del modello 709, nonché il più veloce computer al mondo dell'epoca. Il PDP-1 costava però 120.000 dollari, mentre il modello 7090 ne costava milioni. DEC vendette decine di PDP-1 e questo rappresentò la nascita dell'industria dei microcomputer.

Uno dei primi PDP-1 fu donato al M.I.T., dove attrasse rapidamente l'attenzione di alcuni dei giovani geni in erba, così comuni in quella università. Una delle principali innovazioni del PDP-1 era costituito da un display visuale e dalla capacità di disegnare punti in qualsiasi zona dello schermo da 512 per 512 linee. Da lì a breve gli studenti programmarono la macchina PDP-1 per poter giocare a *spacewar*, facendo così conoscere al mondo il primo videogame della storia.

Pochi anni dopo DEC introdusse il modello PDP-8, una macchina a 12 bit, molto più economica del PDP-1 (16.000 dollari). La principale innovazione introdotta dal PDP-8 fu quella di avere un unico bus, chiamato *omnibus*, mostrato nella Figura 1.6. Un **bus** è un insieme di cavi paralleli utilizzati per connettere i diversi componenti di un computer. Questa architettura fu un grande passo in avanti rispetto alla macchina IAS, che era centralizzata rispetto alla memoria, e fu adottata da quasi tutti i successivi computer di piccole dimensioni. DEC riuscì a vendere 50.000 PDP-8, affermandosi così leader nel mercato dei microcomputer.

Nel frattempo la reazione di IBM all'avvento del transistor fu la costruzione del modello 7090, una versione a transistor del computer 709, citato precedentemente, e in seguito del modello 7094. Il computer 7094 aveva un ciclo di clock di 2 microsecondi e possedeva una memoria centrale di 32.536 parole da 36 bit. I modelli 7090 e 7094 segnarono la fine delle macchine in stile ENIAC e dominarono il mondo dell'elaborazione scientifica durante gli anni '60.

Nello stesso momento in cui, grazie al modello 7094, IBM diventava una delle principali forze del mercato, otteneva enormi profitti vendendo una piccola macchina per le aziende, chiamata 1401. Questa macchina era in grado di leggere e scrivere nastri magnetici, leggere e perforare schede e stampare output a una velocità paragonabile a quella del modello 7094, il tutto a una frazione del prezzo. Era quindi perfetta per la gestione dei dati aziendali, ma aveva pessime prestazioni nei calcoli scientifici.

L'architettura del modello 1401 era insolita, dato che non aveva alcun registro e neppure una lunghezza fissa di parola. La sua memoria era composta da 4000 byte da 8 bit, sebbene modelli successivi supportassero fino a 16.000 byte, una dimensione che all'epoca era

Figura 1.6 L'omnibus del computer PDP-8.

stupefacente. Ciascun byte conteneva un carattere da 6 bit, un bit di amministrazione e un bit usato per segnalare la fine di una parola. Per fare un esempio, un'istruzione MOVE specificava un indirizzo d'origine e uno di destinazione e iniziava a spostare byte dall'origine alla destinazione finché non ne incontrava uno il cui bit di fine parola era impostato a 1.

Nel 1964 una piccola e sconosciuta società, la Control Data Corporation (CDC), introdusse il modello 6600, una macchina che era quasi un ordine di grandezza più veloce dell'imponente 7094 e di ogni altra macchina dell'epoca. Per i matematici che si occupavano di analisi numerica, spesso chiamati *number cruncher*, "masticatori di numeri", fu amore a prima vista e il computer CDC fu lanciato verso il successo. Il segreto della sua velocità, e il motivo per cui fosse molto più del modello 7094, stava nel fatto che la CPU era, al suo interno, una macchina altamente parallela. Era dotata di varie unità funzionali che potevano lavorare contemporaneamente: alcune erano destinate a svolgere addizioni, altre moltiplicazioni e altre ancora divisioni. Anche se per ottenere il maggior guadagno possibile era richiesta un'attenta programmazione, con un po' di lavoro si riusciva a far eseguire contemporaneamente anche 10 istruzioni.

Come se non bastasse, il modello 6600 aveva al suo interno un certo numero di piccoli computer che lo aiutavano; un po' come i Sette Nani con Biancaneve, questo voleva dire che la CPU poteva spendere tutto il suo tempo a macinare numeri lasciando ai piccoli computer tutti i dettagli della gestione dei programmi e dell'input/output. A posteriori si può affermare che il 6600 era in anticipo di decenni sui tempi; molte delle idee chiave proprie dei computer moderni possono infatti essere ricondotte a questa macchina.

Il progettista del computer 6600, Seymour Cray, fu una figura leggendaria del livello di von Neumann. Dedicò la sua intera vita a costruire macchine sempre più veloci, ora chiamate **supercomputer**, tra cui gli elaboratori 6600, 7600 e Cray-1. Inventò inoltre un famoso algoritmo per l'acquisto di un'autovettura: si va dal rivenditore più vicino a casa propria, si indica la macchina più vicina alla porta e si dice: "Voglio questa". L'algoritmo impiega il minor tempo possibile per le cose non importanti (come comprare una macchina) per lasciare il maggior tempo possibile alle cose importanti (come progettare supercomputer).

Nella stessa epoca ci furono molti altri computer, ma uno si distinse per un motivo particolare e merita per questo di essere menzionato: il Burroughs B5000. I progettisti di macchine come i modelli PDP-1, 7094 e 6600 si concentravano esclusivamente sull'hardware, sia nel tentativo di renderlo economico (DEC) sia in quello di renderlo veloce (IBM e CDC), mentre ignoravano quasi completamente il software. I progettisti del B5000 adottarono invece un approccio diverso. Costruirono una macchina con il preciso intento di programmarla in Algol 60, un precursore di C e Java, e inclusero nell'hardware molte caratteristiche finalizzate a facilitare il compito del compilatore. Nacque così l'idea che anche il software ha la sua importanza; purtroppo questa considerazione fu quasi subito abbandonata.

1.2.4 Terza generazione – Circuiti integrati (1965-1980)

Nel 1958 l'invenzione dei circuiti integrati su silicio da parte di Robert Noyce permise di realizzare su un unico chip decine di transistor. Questo metodo di assemblaggio rese possibile la costruzione di computer più piccoli, più veloci e più economici rispetto ai loro predecessori basati su transistor. Alcuni dei computer più significativi di questa generazione sono descritti in seguito.

Fin dal 1964 IBM era la società leader nel mondo dei computer e si trovava di fronte a un grande problema che riguardava le sue due macchine di successo, i modelli 7094 e 1401: essi erano totalmente incompatibili. Uno era un "macinatore di numeri" a grande velocità che usava aritmetica binaria parallela su registri a 36 bit, mentre l'altro era un popolare processore di input/output che usava aritmetica decimale su parole di memoria a lunghezza variabile. Molte fra le società clienti di IBM avevano comprato entrambe le macchine e non amavano l'idea di dover avere due dipartimenti di programmazione separati senza nulla in comune.

Quando arrivò il momento di sostituire questi modelli, IBM fece un cambio radicale. Introdusse un'unica linea di prodotti, il System/360, basata su circuiti integrati e progettata sia per i calcoli scientifici sia per quelli commerciali. Il System/360 presentava molteplici innovazioni, ma la più importante fu che si trattava di una famiglia di circa mezza dozzina di macchine dotate dello stesso linguaggio assemblativo, ma di dimensione e potenza crescenti. Una società poteva quindi sostituire il proprio modello 1401 con un 360 Model 30 e il proprio modello 7094 con un 360 Model 75. Il Model 75 era il più grande e il più veloce (e anche il più caro), ma il software scritto per uno di questi modelli poteva, in linea di principio, essere eseguito anche su uno differente. In pratica il software scritto per un piccolo modello funzionava senza problemi su un modello più grande, ma quando ci si spostava verso un modello di dimensioni inferiori il programma poteva non entrare in memoria. In ogni caso ciò rappresentava un grande avanzamento rispetto alla situazione delle macchine 7094 e 1401. L'idea di avere una *famiglia di computer* fu subito colta anche da molti altri produttori che, in pochi anni, misero sul mercato linee di macchine che ricoprivano un grande spettro di prezzi e prestazioni. La Figura 1.7 mostra alcune caratteristiche della famiglia 360 originaria (successivamente vennero introdotti anche altri modelli).

Un'altra grande innovazione della serie 360 fu la **multiprogrammazione**, grazie a cui è possibile avere più programmi in memoria allo stesso tempo, di modo che quando si è in attesa di completare un'operazione di input/output si possono eseguire dei calcoli. Il risultato che si ottiene con questa tecnica è un maggior utilizzo della CPU.

Il 360 fu anche il primo a essere capace di emulare (simulare) altri computer. Il modello più piccolo poteva emulare il computer 1401, mentre quelli più grandi erano in grado di emulare il modello 7094, di modo che, durante il passaggio ai computer 360, i clienti potessero continuare a utilizzare i loro vecchi programmi (binari) senza modificarli. Alcuni modelli eseguivano i programmi scritti per il 1401 in modo talmente più veloce che molti clienti non convertirono mai il proprio software.

Proprietà	Model 30	Model 40	Model 50	Model 65
Prestazioni relative	1	3,5	10	21
Ciclo di clock (in miliardesimi di secondo)	1000	625	500	250
Quantità massima di memoria (in byte)	65.536	262.144	262.144	524.288
Numero di byte prelevati per ciclo di clock	1	2	4	16
Massimo numero di canali per i dati	3	3	4	6

Figura 1.7 L'offerta iniziale della linea di prodotti IBM 360.

Sul 360 l'emulazione risultava facile, in quanto tutti i modelli iniziali, e molti di quelli successivi, erano microprogrammati. Tutto quello che IBM doveva fare era scrivere tre microprogrammi, uno per l'insieme d'istruzioni native del modello 360, uno per quelle del modello 1401 e uno per quelle del modello 7094. Questa flessibilità fu una delle ragioni principali per cui fu introdotta la microprogrammazione.

Il modello 360, grazie a un compromesso, risolse anche il dilemma dell'aritmetica binaria parallela e di quella decimale seriale: la macchina aveva 16 registri da 32 bit per l'aritmetica binaria, ma la sua memoria era orientata al byte, come quella del modello 1401. Era inoltre dotata d'istruzioni seriali simili a quelle del modello 1401 per lo spostamento in memoria di blocchi di dimensioni variabili.

Un'altra caratteristica saliente del computer 360 era un enorme (per l'epoca) spazio degli indirizzi di 2^{24}=16.777.216 byte. All'epoca, una tale quantità di memoria appariva praticamente come infinita, dato che il costo della memoria era di qualche dollaro al byte. Sfortunatamente la serie 360 fu seguita dalla serie 370, dalla serie 4300, dalla serie 3080 e dalla serie 3090, tutte basate sulla stessa architettura. A metà degli anni '80 il limite della memoria divenne un problema reale e IBM fu costretta ad abbandonare in parte la compatibilità quando introdusse gli indirizzi a 32 bit, necessari per indirizzare una nuova memoria di 2^{32} byte.

Con il senno di poi si potrebbe discutere sul perché non si fosse pensato fin dall'inizio all'utilizzo di indirizzi a 32 bit, dato che le macchine erano state dotate di registri e parole a 32 bit; all'epoca però nessuno avrebbe mai potuto immaginare una macchina con 16 milioni di byte di memoria. Accusare IBM per questa mancanza di previsione sarebbe come accusare gli attuali produttori di computer di fornire indirizzi limitati a 32 bit. Fra pochi anni i personal computer avranno bisogno di più di 4 miliardi di byte di memoria e, a quel punto, anche gli indirizzi a 32 bit diventeranno intollerabilmente piccoli.

Nella terza generazione, anche il mondo dei microcomputer fece un gran passo in avanti con l'introduzione da parte di DEC della serie PDP-11, un successore a 16 bit del modello PDP-8. Sotto molti punti di vista la serie PDP-11 era una sorta di piccolo fratello della serie 360 così come il computer PDP-1 lo era stato per il modello 7094. Entrambi i computer 360 e PDP-11 avevano i registri orientati alla parola e la memoria orientata al byte, e tutti e due vennero prodotti in una gamma in cui il rapporto prezzo/prestazioni variava. Il modello PDP-11 ebbe un enorme successo specialmente presso le università, permettendo a DEC di mantenere la leadership sugli altri produttori di microcomputer[2].

1.2.5 Quarta generazione – Integrazione a grandissima scala (1980-?)

Negli anni '80 la tecnologia **VLSI** (*Very Large Scale Integration*, "integrazione a larghissima scala") permise di inserire in un unico chip prima decine di migliaia, poi centinaia di migliaia e infine milioni di transistor. Questo sviluppo portò velocemente alla realizzazione di computer più piccoli e più veloci. Prima del PDP-1 i computer erano talmente grandi e costosi che, per farli funzionare, aziende e università dovevano disporre di speciali uffici chiamati **centri di calcolo**, mentre, con l'avvento dei microcomputer, ogni dipartimento poteva dotarsi di un

[2] Questi modelli erano generalmente chiamati *minicomputer* (*N.d.Rev.*).

proprio elaboratore. Dal 1980 i prezzi crollarono al punto che anche i privati potevano permettersi di possedere un computer: ebbe così inizio l'era del personal computer.

I personal computer furono utilizzati in modo molto diverso rispetto ai computer di grandi dimensioni. Vennero usati per l'elaborazione di testi, per l'amministrazione domestica (con i ben noti *spreadsheet*) e per numerose altre applicazioni fortemente interattive (come i videogiochi) che computer di dimensioni maggiori non potevano trattare in modo altrettanto pratico.

I primi personal computer venivano generalmente venduti in scatole di montaggio. Ciascun kit conteneva una scheda con circuito stampato, vari chip, tra cui di solito vi era un Intel 8080, alcuni cavi, un trasformatore e talvolta un floppy disk da 8 pollici. Era compito dell'acquirente montare le parti per costruire il computer, e il software non era incluso: se si voleva un qualsiasi programma occorreva scriverlo. Successivamente si diffuse per i processori 8080, il sistema operativo CP/M, scritto da Gary Kildall. Si trattava di un vero sistema operativo (su floppy disk), con un file system e una serie d'istruzioni che l'utente doveva scrivere con la tastiera per farle leggere da un interprete di comandi (*shell*).

Un altro dei primi personal computer fu l'Apple e in seguito l'Apple II, progettati da Steve Jobs e Steve Wozniak nel loro famoso garage. Questa macchina ebbe un'enorme diffusione tra i privati e nelle scuole, rendendo la Apple dall'oggi al domani un serio concorrente sul mercato.

IBM, la forza principale dell'industria dei computer, dopo aver osservato a lungo e considerato a fondo che cosa stavano facendo le aziende concorrenti, decise infine di entrare nel mercato dei personal computer. Dato che progettare da zero una nuova macchina usando solo componenti proprietarie avrebbe richiesto troppo tempo, IBM scelse una strada alternativa e piuttosto inusuale. Diede a un suo dirigente, Philip Extridge, una grande borsa carica di soldi e gli affidò l'incarico di allontanarsi dalle interferenze dei burocrati della sede centrale di Armonk, nello stato di New York, e di non tornare se non con un personal computer funzionante. Estridge iniziò a fare acquisti a Boca Raton, in Florida, lontano dalla sede centrale, scegliendo come CPU l'Intel 8088 e costruendo così il Personal Computer IBM utilizzando componenti commerciali. Il PC IBM fu introdotto nel mercato nel 1981 e divenne subito il computer più venduto nella storia.

IBM fece anche qualcos'altro di inusuale di cui in seguito si sarebbe pentita. Diversamente dal solito, invece di mantenere completamente segreto il progetto della macchina (o almeno di proteggerlo con un muro di brevetti) pubblicò, in un libro venduto a soli 49 dollari, tutti gli schemi al completo, compresi i diagrammi dei circuiti. L'idea era quella di permettere ad altre società di realizzare delle schede aggiuntive per il PC IBM, in modo da aumentarne la flessibilità e la popolarità. Sfortunatamente per IBM numerose altre società cominciarono a realizzare dei **cloni** del PC, dato che il progetto era completamente pubblico e i componenti facilmente reperibili sul mercato. Spesso questi computer erano molto più economici di quelli IBM; in questo modo prese il via una nuova industria.

Nonostante altre compagnie, tra cui Commodore, Apple e Atari, realizzassero personal computer senza utilizzare le CPU Intel, la forza del mercato dei PC IBM era talmente grande che gli altri ne vennero schiacciati. Solo in pochi sopravvissero e soltanto in mercati di nicchia.

Uno dei sopravvissuti, seppur a mala pena, fu il Macintosh di Apple. Il Macintosh era stato introdotto nel 1984 come successore dello sfortunato Apple Lisa, il primo computer dotato di una **GUI** (*Graphical User Interface*, "interfaccia grafica per l'utente"), simile alla ormai

popolare interfaccia di Windows. Lisa fallì a causa del costo troppo elevato, mentre il più economico Macintosh, introdotto l'anno successivo, riscosse un enorme successo e ispirò amore e passione nei suoi numerosi ammiratori.

Il giovane mercato dei personal computer fece nascere anche il desiderio, fino ad allora senza precedenti, di avere computer portatili. A quel tempo un computer portatile aveva senso quanto può averne oggi un frigorifero portatile. Il primo vero personal computer portatile fu l'Osborne-1, che con i suoi 11 Kg fu più che altro un "computer da bagagliaio". Eppure, dimostrò che era possibile realizzare computer portatili. L'Osborne-1 ebbe un successo commerciale modesto, ma un anno dopo Compaq produsse il suo primo clone di PC IBM portatile e divenne rapidamente il leader di tale mercato.

La versione iniziale del PC IBM veniva venduta insieme al sistema operativo MS-DOS, fornito dalla Microsoft Corporation, una società che all'epoca era ancora piccola. Visto che Intel riusciva a produrre CPU sempre più potenti, IBM e Microsoft poterono sviluppare un successore di MS-DOS, chiamato OS/2, dotato di un'interfaccia utente grafica simile a quella dell'Apple Macintosh. Allo stesso tempo Microsoft, pensando all'ipotesi che OS/2 potesse non prendere piede, cominciò a sviluppare un suo proprio sistema operativo, Windows, eseguito sopra MS-DOS. Per riassumere questa lunga vicenda basti dire che OS/2 non ebbe successo, che fra IBM e Microsoft vi fu un'accesa e pubblica disputa e che Microsoft continuò per la propria strada, facendo di Windows un enorme successo. Come la piccola Intel e la ancora più piccola Microsoft abbiano potuto detronizzare IBM, una delle più grandi, ricche e potenti società nella storia del mondo, è senza dubbio una lezione che viene attentamente studiata in tutte le scuole di economia del mondo.

Grazie al successo conseguito con l'8088, Intel continuò a realizzarne versioni via via migliori e più potenti. Particolarmente degno di nota fu il 386, introdotto nel 1985, in pratica il primo Pentium. Gli attuali Pentium infatti, sebbene siano molto più veloci, dal punto di vista dell'architettura non sono che una versione "truccata" del 386.

Alla metà degli anni '80 iniziò ad affermarsi un nuovo tipo di progettazione chiamata RISC, che consisteva nel sostituire le complicate architetture esistenti con altre molto più semplici (e più veloci). Queste macchine erano in grado di eseguire più istruzioni allo stesso tempo, spesso in un ordine diverso da quello in cui apparivano nel programma. I concetti di CISC, RISC e superscalare saranno introdotti nel Capitolo 2 e verranno analizzati approfonditamente nel corso del libro.

I computer restarono a 8, 16 o 32 bit fino al 1992, quando DEC presentò il rivoluzionario Alpha, una vera macchina a 64 bit di tipo RISC che surclassava di gran lunga le prestazioni degli altri computer. La macchina ebbe un successo limitato, ma dovette passare un decennio prima che le macchine a 64 bit cominciassero ad affermarsi su grande scala, soprattutto fra i server di fascia alta.

1.2.6 Quinta generazione – Computer invisibili

Nel 1981 il governo giapponese annunciò di voler stanziare 500 milioni di dollari per aiutare le società locali nella costruzione della quinta generazione di computer, che si sarebbe basata sull'intelligenza artificiale e che avrebbe rappresentato un enorme balzo in avanti rispetto alla "stupida" quarta generazione. I produttori di computer americani ed europei, avendo visto le compagnie giapponesi prendere piede in molti mercati, come quello delle telecamere, degli stereo e dei televisori, piombarono improvvisamente nel panico più totale e cominciarono a chiedere, tra le altre cose, finanziamenti ai rispettivi governi. Il progetto giapponese di quinta generazione, nonostante tanta enfasi, sostanzialmente fallì e venne abbandonato nel silenzio. In un certo senso fu come per l'*analytical engine* di Babbage; si trattò di un'idea visionaria talmente avanti rispetto ai tempi che la tecnologia necessaria non era neanche prevedibile.

Quella che può essere definita la quinta generazione di computer vide comunque la luce, seppur in un modo inaspettato: i computer si rimpicciolirono. L'Apple Newton, prodotto nel 1993, mostrò che un computer poteva essere costruito con dimensioni non più grandi di quelle di un lettore di audiocassette. Newton utilizzava come dispositivo di input la scrittura a mano libera, e questo ne rappresentò un grande ostacolo; in seguito però altre macchine della stessa tipologia, ora chiamate **PDA** (*Personal Digital Assistants*, "assistente digitale personale"), hanno migliorato l'interfaccia utente e sono diventate estremamente popolari. Oggi molte di loro hanno la stessa potenza computazionale di computer di pochi anni prima.

Neanche i PDA possono essere considerati veramente rivoluzionari. Molto più importanti sono invece i computer cosiddetti "invisibili", poiché integrati in elettrodomestici, orologi, carte di credito e numerosi altri dispositivi (Bechini et al., 2004). In un ampio spettro di applicazioni questi processori garantiscono grandi funzionalità a basso costo. Anche se si può discutere sul fatto che questi processori siano veramente una nuova generazione (dato che sono in circolazione dagli anni '70), è indubbio che stiano rivoluzionando il funzionamento di migliaia di elettrodomestici e altri dispositivi. Stanno già cominciando ad avere un forte impatto sul mondo e nei prossimi anni la loro influenza crescerà rapidamente. Un aspetto non comune di questi computer integrati è che l'hardware e il software sono spesso **coprogettati** (Henkel et al., 2003). Più avanti nel libro torneremo a occuparci di loro.

Se la prima generazione è rappresentata dalle macchine a valvole (per esempio ENIAC), la seconda dalle macchine a transistor (per esempio, l'IBM 7094), la terza dalle macchine a circuiti integrati (per esempio, l'IBM 360) e la quarta dai personal computer (per esempio, le CPU Intel), la quinta generazione è in realtà caratterizzata più da un cambiamento di modello che da una specifica nuova architettura. In futuro i computer si troveranno ovunque e saranno integrati in ogni oggetto, in poche parole essi saranno invisibili; faranno parte dei comuni gesti della vita di ogni giorno, come aprire porte, accendere luci, spendere denaro e migliaia di altre azioni. Questo modello, ideato dallo scomparso Mark Weiser, venne originariamente chiamato *ubiquitous computing* ("computazione onnipresente"), anche se spesso ci si riferisce a esso con il termine *pervasive computing* ("computazione pervasiva") (Weiser, 2002): cambierà profondamente il mondo, così come lo fece la rivoluzione industriale. Nel libro non tratteremo ulteriormente l'argomento, ma per avere maggiori informazioni si consultino (Lyytinen e Yoo, 2002; Saha e Mukherjee, 2003; Sakamura, 2002).

1.3 Tipologie di computer

Mentre nel paragrafo precedente abbiamo affrontato brevemente la storia dei sistemi di elaborazione, in questo guarderemo al presente e rivolgeremo anche uno sguardo al futuro. Sebbene i personal computer siano gli elaboratori più conosciuti, al giorno d'oggi esistono anche altri tipi di macchine: per questo vale la pena dare un'occhiata a quanto c'è intorno a noi.

1.3.1 Forze tecnologiche ed economiche

L'industria dei computer avanza più velocemente di tutte le altre. La principale forza trainante è la capacità, che i produttori di circuiti hanno, di integrare di giorno in giorno sempre più transistor all'interno dei chip. Avere più transistor, cioè piccoli interruttori elettronici, significa avere memorie più grandi e processori più potenti. Gordon Moore, cofondatore ed ex presidente di Intel, un giorno scherzando disse che, se la tecnologia dell'aviazione fosse cresciuta velocemente quanto quella dei computer, ora un aeroplano costerebbe 500 euro e farebbe il giro della terra in soli 20 minuti con 20 litri di carburante. Per di più un tale aeroplano avrebbe le dimensioni di una scatola da scarpe. In particolare Moore, durante la preparazione di un discorso dinanzi a un gruppo industriale, sottolineò che ogni nuova generazione di chip veniva introdotta 3 anni dopo quella precedente. Dato che ogni nuova generazione ha una quantità di memoria quattro volte maggiore della precedente, comprese che il numero di transistor in un chip aumentava a una velocità costante e predisse che questa crescita sarebbe continuata allo stesso modo per decenni. Questa osservazione è conosciuta come la **legge di Moore**. Attualmente con il termine "legge di Moore" si intende il fatto che il numero di transistor raddoppia ogni 18 mesi; questo è equivalente a dire che il numero di transistor aumenterebbe circa del 60% ogni anno. La Figura 1.8, che riporta le dimensioni di chip di memoria e la loro data di apparizione, conferma che la legge di Moore è rimasta valida per oltre tre decenni.

Ovviamente la legge di Moore non è una vera e propria legge, ma un'osservazione empirica sulla velocità con cui i fisici dello stato solido e gli ingegneri hanno fatto avanzare lo

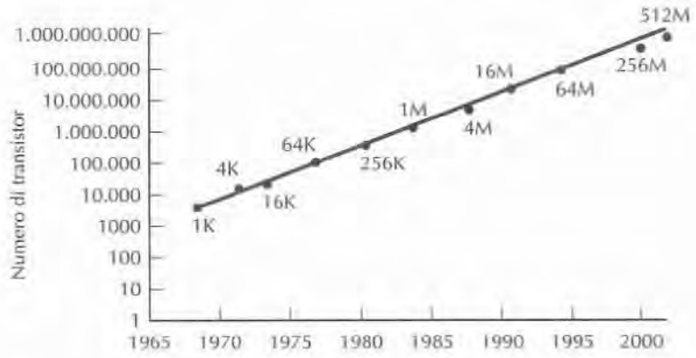


Figura 1.8 La legge di Moore prevede un aumento annuale del 60% del numero di transistor che può essere inserito su un chip. I valori presenti vicino ai punti del grafico rappresentano le dimensioni della memoria, in bit.

stato dell'arte, e una previsione che ciò continuerà anche in futuro. Alcuni osservatori del mondo dell'industria si aspettano che la legge di Moore continui per almeno un altro decennio. A quel punto però i transistor saranno costituiti da un numero troppo basso di atomi per poter essere ancora affidabili; ciononostante gli avanzamenti nel campo della computazione quantistica potrebbero far immaginare cambiamenti simili a quelli presentati in Oskin et al., 2002. Tuttavia altri osservatori si aspettano che la dissipazione di energia, la dispersione di corrente e altri effetti simili smentiranno ancor prima la legge di Moore e provocheranno seri problemi che in qualche modo dovranno essere risolti (Bose, 2004; Kim et al., 2003).

La legge di Moore ha creato quello che gli economisti chiamano un **ciclo virtuoso**. Gli avanzamenti tecnologici (transistor/chip) portano a prodotti migliori e allo stesso tempo più economici. Prezzi inferiori portano a loro volta alla creazione di nuove applicazioni (nessuno programmava videogiochi per computer quando un elaboratore costava 10 milioni di dollari). Nuove applicazioni creano nuovi mercati e fanno sorgere nuove società che cercano di trarne vantaggio. L'esistenza di tutte queste società genera competizione, che, a sua volta, crea la domanda economica per avere tecnologie migliori grazie a cui sconfiggere la concorrenza. In questo modo il cerchio si chiude.

Un altro fattore che guida lo sviluppo tecnologico è la prima legge del software di Nathan (dovuta a Nathan Myhrvold, un ex dirigente di Microsoft). Egli affermò che: "Il software è come un gas. Si espande fino a riempire il recipiente in cui è contenuto". Tornando agli anni '80 l'elaborazione di testi era fatta mediante programmi come *troff*. *Troff* occupa in memoria uno spazio dell'ordine dei KB, mentre i moderni elaboratori di testo occupano vari MB; senza alcun dubbio quelli futuri richiederanno GB di memoria (in prima approssimazione i prefissi K, M e G significano rispettivamente migliaia, milioni e miliardi, si veda comunque il Paragrafo 1.5 per maggiori dettagli). Il software, continuando ad acquisire sempre più funzionalità (come crostacei che si attaccano via via alla chiglia di una barca), crea una richiesta costantemente crescente di processori più veloci, memorie più grandi e supporti di I/O di maggiore capacità.

Se nel corso degli anni l'aumento del numero di transistor per chip è stato sensazionale, i progressi delle altre tecnologie legate ai computer non sono stati da meno. Per fare un esempio, il PC IBM/XT è stato introdotto nel 1982 con un hard disk da 10 MB, mentre a distanza di vent'anni i suoi successori sono comunemente dotati di hard disk da 100 GB. Questo aumento della capacità di quattro ordini di grandezza in 20 anni rappresenta un incremento annuale del 58%. Misurare i progressi dei dischi è però un compito complesso, dato che oltre alla dimensione vi sono altri parametri da considerare, come la velocità di trasferimento, il tempo di ricerca e il prezzo. In ogni caso con quasi tutte queste metriche si verifica che dal 1982 il rapporto prezzo/prestazioni è migliorato ogni anno almeno del 50%. Questo enorme miglioramento delle prestazioni dei dischi, associato al fatto che il volume economico degli hard disk prodotti nella Silicon Valley ha superato quello delle CPU, ha spinto Al Hoagland ad affermare che il nome della famosa valle è sbagliato: andrebbe infatti chiamata Iron Oxide Valley (essendo questo il materiale – un ossido di ferro – su cui vengono registrati i dati nei dischi).

Un'altra area ad aver conosciuto uno spettacolare progresso è stata quella delle telecomunicazioni e delle reti. In meno di due decenni siamo passati dai modem a 300 bit/s, ai modem analogici a 56.000 bit/s fino alle fibre ottiche a 10^{12} bit/s. I cavi telefonici transatlantici in fibra ottica, come il TAT-12/13, costano circa 700 milioni di euro, durano

10 anni, e possono trasportare 300.000 chiamate simultanee; questi dati corrispondono a meno di un centesimo ogni 10 minuti di chiamata intercontinentale. È già stato dimostrato che si possono realizzare sistemi ottici di comunicazione a 10^{12} bit/s su distanze superiori a 100 km e senza l'utilizzo di amplificatori. La crescita esponenziale di Internet non richiede ulteriori commenti.

1.3.2 Spettro di computer

Richard Hamming, ex-ricercatore dei Bell Labs, una volta osservò che un cambiamento della quantità di un ordine di grandezza produce un cambiamento della qualità. Secondo questa osservazione una macchina da corsa che può raggiungere i 1000 Km/h nel deserto del Nevada è un tipo di vettura fondamentalmente diverso da una comune automobile che va a 100 Km/h su un'autostrada. Analogamente un grattacielo da 100 piani non è semplicemente una versione ingrandita di un condominio da 10 piani. Nel caso dei computer non si parla di un fattore 10, ma addirittura di un fattore 1 milione nell'arco di tre decenni.

Il guadagno fornito dalla legge di Moore può essere sfruttato in modi diversi. Uno di loro consiste nel costruire computer sempre più potenti allo stesso prezzo; l'altro nel costruire di anno in anno sempre lo stesso computer, ma a un prezzo inferiore. L'industria dei computer ha seguito entrambi gli approcci, e altri ancora, al punto che oggi esiste un'ampia varietà di computer. La Figura 1.9 mostra una classificazione sommaria dei computer attuali.

Nei paragrafi seguenti esamineremo ciascuna di queste categorie e ne considereremo brevemente le proprietà.

Tipo	Prezzo ($)	Esempio di applicazione
Computer usa e getta	0,5	Cartoline d'auguri
Microcontrollore	5	Orologi, automobili, elettrodomestici
Computer da gioco	50	Videogiochi
Personal computer	500	Computer desktop o portatili
Server	5K	Server di rete
Raggruppamento di workstation	50-500K	Minisupercomputer di un dipartimento
Mainframe	5M	Elaborazione batch dei dati in una banca

Figura 1.9 L'attuale spettro di computer disponibili. I prezzi vanno interpretati "cum granu salis" (o meglio ancora, "cum saxo salis").

1.3.3 Computer usa e getta

Al gradino più basso troviamo i chip inseriti all'interno di cartoline d'auguri, che emettono la melodia di "Happy Birthday to You" o qualche altro motivetto altrettanto popolare. L'Autore non ha ancora trovato una cartolina di condoglianze che suoni un canto funebre, ma ora che ha reso pubblica questa idea si aspetta di vederne apparire presto una sul mercato. Per chi è cresciuto con grandi computer da milioni di dollari l'idea di un computer usa e getta ha senso quanto può averne quella di un aereo con lo stesso utilizzo.

Tuttavia i computer usa e getta fanno parte del nostro mondo e continueranno a farne parte anche in futuro. Probabilmente il più importante sviluppo nell'area di questi tipi di computer è rappresentato dai chip **RFID** (*Radio Frequency Identification*, "identificazione a radiofrequenza"). Oggi è possibile costruire, per pochi centesimi, chip RFID senza batteria, larghi meno di 0,5 mm, caratterizzati da un numero a 128 bit predefinito e univoco e contenenti al loro interno un piccolo radiotrasmettitore. Quando ricevono impulsi radio da un'antenna esterna i chip RFID si caricano grazie al segnale entrante sufficientemente a lungo da riuscire a ritrasmettere all'antenna il proprio numero identificativo. Anche se questi chip sono piccoli, le loro possibili implicazioni non lo sono affatto.

Partiamo dalla banale applicazione di eliminare il codice a barre dai prodotti. Sono già stati realizzati test sperimentali in cui i prodotti venduti in un negozio hanno, al posto del codice a barre, un chip RFID applicato dal produttore. Il cliente sceglie i prodotti desiderati, li ripone in un carrello della spesa e lo spinge semplicemente fuori dal negozio. All'uscita un lettore dotato d'antenna spedisce un segnale richiedendo a ciascun prodotto di identificarsi, cosa che viene fatta attraverso una breve trasmissione radio. Anche il cliente viene identificato per mezzo di un chip presente sui propri bancomat o carta di credito, così che alla fine del mese il negozio potrà inviargli una dettagliata ricevuta di tutti gli acquisti effettuati. Se il cliente non ha un conto in una banca o una carta di credito che supporta RFID, scatta invece un allarme. Questo sistema non solo può sostituire i cassieri ed eliminare le relative attese in coda, ma può anche servire da sistema antifurto, dato che non è di alcuna utilità nascondere un prodotto in tasca o nello zainetto.

Una proprietà interessante di questo sistema è che, mentre il codice a barre identifica solo il tipo di prodotto, i chip RFID, grazie ai 128 bit a disposizione, possono identificare anche il prodotto stesso. Ne consegue che, per esempio, ogni pacchetto di aspirine presente sullo scaffale di un supermercato potrebbe avere un diverso codice RFID. Questo significa che se un produttore farmaceutico dovesse scoprire un difetto in un lotto di aspirine già distribuito, si potrebbe richiedere ai supermercati di tutto il mondo di far suonare un allarme se un cliente compra un pacchetto il cui numero RFID fa parte di tale gruppo. Ciò sarebbe attuabile anche se l'acquisto fosse effettuato in un paese lontano e a distanza di mesi; le aspirine non appartenenti al lotto difettato non provocherebbero invece alcun allarme.

Ma etichettare pacchetti di aspirine, biscotti o cibo per cani è soltanto l'inizio. Perché fermarsi a etichettare i biscotti per cani quando si può etichettare il cane stesso? Alcuni padroni stanno già chiedendo ai veterinari di impiantare nei loro animali un chip RFID, in modo da poterli rintracciare nel caso venissero rubati o si perdessero. Anche gli allevatori vogliono etichettare il loro bestiame. Il passo successivo che ci si aspetta è quello di genitori nervosi che richiedono al pediatra di impiantare un chip RFID nei loro figli, per poterli ritrovare nel caso venissero rapiti o si perdessero. Già che ci siamo, perché non obbligare gli ospedali a impiantarli in tutti i neonati, in modo da evitare gli scambi di bebè? Senza dubbio i governi e la polizia possono trovare ottime ragioni per identificare e tracciare i movimenti di tutti i cittadini in qualsiasi momento. Detto questo, le "implicazioni" dei chip RFID a cui si alludeva precedentemente risultano un po' più chiare.

Un'altra applicazione (leggermente meno controversa) dei chip RFID è l'identificazione di veicoli. Quando un treno, con chip RFID integrati, passa vicino a un lettore, il computer che vi è collegato ottiene una lista delle vetture che sono transitate. Questo sistema permette di localizzare agevolmente tutti i vagoni, il che può essere d'aiuto a forni-

tori, clienti e ferrovie. Uno schema simile sarebbe applicabile anche ai camion, mentre per le automobili l'idea potrebbe essere usata per il pagamento dei pedaggi autostradali per via elettronica.

Anche la gestione dei bagagli delle compagnie aeree, così come molti altri sistemi di consegna, potrebbero trarre vantaggio dai chip RFID. All'areoporto di Heathrow, a Londra, è stato condotto un esperimento per sollevare i passeggeri in arrivo dal ritiro dei loro bagagli. Le borse dei passeggeri che avevano acquistato questo servizio venivano marcate mediante chip RFID, poi spedite separatamente all'interno dell'areoporto e infine consegnate direttamente nei rispettivi hotel. Fra le svariate possibilità di utilizzo dei chip RFID si possono immaginare macchine che, giunte alla stazione di verniciatura di una linea di assemblaggio, comunicano il colore che deve essere loro applicato; ulteriori possibili impieghi comprendono lo studio delle migrazioni di animali, la produzione di vestiti che comunicano alla lavatrice la temperatura da utilizzare e molto altro ancora. Alcuni chip potrebbero essere integrati con sensori in modo che i bit inferiori del loro numero identificativo potrebbero specificare temperatura corrente, pressione, umidità o altre variabili ambientali.

Chip RFID più avanzati sono in grado di memorizzare dati in modo permanente. Questa possibilità ha spinto la Banca Centrale Europea a decidere di inserire tali chip nelle banconote di euro dei prossimi anni. I chip sarebbero in grado di memorizzare il loro percorso. Ciò non solo renderebbe virtualmente impossibile contraffare le banconote di euro, ma permetterebbe anche di individuare i riscatti pagati ai rapitori e i bottini delle rapine; inoltre sarebbe facile tener traccia del denaro sporco e invalidarne le banconote. In futuro, quando il denaro cesserà di essere anonimo, una procedura standard di polizia potrebbe essere quella di controllare dove sia stato di recente il denaro di un sospetto. Chi avrà bisogno di impiantare chip nelle persone quando i loro portafogli ne saranno già pieni? Molto probabilmente, quando l'opinione pubblica capirà che cosa permettono di fare i chip RFID, nasceranno dibattiti pubblici in materia.

La tecnologia utilizzata nei chip RFID si sta sviluppando rapidamente. I più piccoli sono passivi (non alimentati internamente) e capaci soltanto di trasmettere il loro numero univoco quando richiesto. Quelli più grandi invece sono attivi, possono contenere una piccola batteria e un computer elementare, e sono in grado di compiere alcuni calcoli. Le piccole carte utilizzate per le transazioni economiche rientrano in questa categoria.

I chip RFID si distinguono non solo per essere attivi o passivi, ma anche in base allo spettro di radiofrequenze a cui possono rispondere. Quelli che operano a basse frequenze hanno una velocità di trasferimento dati limitata, ma possono essere captati da un'antenna anche a grande distanza. Quelli che operano ad alte frequenze hanno invece una più alta velocità di trasferimento dati, ma un raggio d'azione più ristretto. I chip differiscono anche sotto altri aspetti e continuano a essere migliorati. Internet è piena d'informazioni riguardanti i chip RFID; un buon punto di partenza è il sito www.rfid.org.

1.3.4 Microcontrollori

Al gradino successivo troviamo i computer integrati in apparecchiature che non sono vendute come elaboratori. I computer integrati in un dispositivo, a volte chiamati **microcontrollori**, lo comandano e ne gestiscono l'interfaccia utente. I microcontrollori sono presenti in un gran numero di apparecchi molto diversi fra loro; in seguito sono elencate alcune di queste categorie:

1. elettrodomestici (radiosveglie, lavatrici, asciugatrici, forni a microonde, impianti antifurto)
2. dispositivi per la comunicazione (telefoni senza fili, cellulari, fax, cercapersone)
3. periferiche del computer (stampanti, scanner, modem, lettori CD-ROM)
4. apparecchi per l'intrattenimento (videoregistratori, DVD, stereo, lettori MP3, decoder video)
5. dispositivi per le immagini (TV, macchine fotografiche digitali, videocamere digitali, obiettivi, fotocopiatrici)
6. strumenti medicali (raggi X, MRI, elettrocardiogrammi, termometri digitali)
7. armi (missili, torpedini)
8. apparecchiature per gli acquisti (macchinette per la distribuzione automatica, bancomat, registratori di cassa)
9. giochi (bambole parlanti, console per videogame, auto o barche telecomandate).

Un'automobile di fascia alta può facilmente contenere 50 microcontrollori che comandano diversi sottosistemi tra cui l'antibloccaggio dei freni, l'iniezione della benzina, la radio e il GPS. Un jet ne può contenere anche più di 200, mentre una famiglia può facilmente possederne centinaia senza neanche saperlo. Fra pochi anni praticamente ogni dispositivo elettrico conterrà un microcontrollore. Il numero di microcontrollori venduti ogni anno surclassa di vari ordini di grandezza le vendite di tutti gli altri tipi di computer, fatta eccezione per quelli usa e getta.

Se i chip RFID sono sistemi minimali, i microcontrollori sono invece sistemi completi, anche se di piccole dimensioni. Ciascun microcontrollore è dotato di un processore, di una memoria e di capacità di I/O. Queste ultime di solito comprendono la gestione di pulsanti e interruttori del dispositivo e il controllo di luci, display e audio. Nella maggior parte dei casi il software è integrato nel chip sotto forma di una memoria di sola lettura creata durante la fabbricazione del microcontrollore. Generalmente i microcontrollori possono essere di due tipi: a uso generale, o specifico per un'applicazione. I primi non sono altro che piccoli, ma comuni, computer; mentre gli ultimi hanno un'architettura e un insieme d'istruzioni progettati specificatamente per una particolare applicazione, per esempio multimediale. I microcontrollori esistono in versioni da 4, 8, 16 e 32 bit.

Tuttavia anche i microcontrollori di uso generale differiscono per vari aspetti dai PC standard. Prima di tutto sono estremamente critici dal punto di vista dei costi. Un'azienda che deve comprare milioni di microcontrollori potrebbe basare la propria scelta su una differenza di prezzo di un centesimo per unità. Questo vincolo porta i produttori di microcontrollori a basare le proprie scelte architetturali molto più sui costi di produzione, rispetto ai produttori di chip da centinaia di euro. Il prezzo dei microcontrollori varia tuttavia in modo rilevante a seconda del loro numero di bit, della grandezza e tipo della memoria di cui sono dotati e di altri fattori; per farsi un'idea, un microcontrollore a 8 bit acquistato in grandi quantità può costare meno di 10 centesimi. Questo prezzo rende possibile l'inserimento di un computer all'interno di una radiosveglia da 9,95 euro.

In secondo luogo, praticamente tutti i microcontrollori lavorano in tempo reale; essi ricevono uno stimolo e ci si aspetta che diano una risposta immediata. Una comune situazione

è l'accensione di una luce quando l'utente preme un pulsante; in tal caso non ci deve essere alcun ritardo tra il momento in cui il pulsante viene premuto e quello in cui la luce si accende. Spesso la necessità di lavorare in tempo reale influisce sul tipo di architettura.

In terzo luogo, i sistemi integrati sono generalmente soggetti a vincoli fisici legati a dimensioni, peso, consumo della batteria e ad altri limiti elettrici e meccanici. I microcontrollori al loro interno devono essere progettati tenendo ben presenti queste restrizioni.

1.3.5 Computer per giocare

A un gradino più in alto si trovano i videogiochi. Si tratta di normali computer dotati di speciali capacità grafiche e sonore, ma poco espandibili e forniti di software limitato. Inizialmente erano basati su CPU a basso costo per poter giocare, tramite il televisore, a semplici giochi d'azione come il ping pong. Negli anni si sono evoluti in sistemi molto più potenti, rivaleggiando con i personal computer sotto determinati aspetti e superandone talvolta le prestazioni.

Per avere un'idea di che cosa ci sia all'interno di una console da gioco consideriamo le specifiche tecniche di tre diffusi prodotti. Come prima macchina, esaminiamo la PlayStation 2 della Sony che contiene una CPU proprietaria (chiamata Emotion Engine) a 128 bit a 295 Mhz, basata sulla CPU MIPS IV di tipo RISC. La PlayStation 2 è dotata inoltre di 32 MB di RAM, un chip grafico dedicato a 160 MHz, un chip audio dedicato a 48 canali e un lettore DVD. Come seconda console, consideriamo la Microsoft XBOX. Questa contiene un Intel Pentium III a 733 MHz con 64 MB di RAM, un chip grafico dedicato a 300 MHz, un chip audio dedicato a 256 canali, un lettore DVD e un hard disk da 8 GB. Infine, prendiamo in esame il Nintendo GameCube che contiene una speciale CPU (chiamata Gekko) a 32 bit e a 485 MHz, derivata dalla CPU RISC PowerPC di IBM, 24 MB di RAM, un chip grafico dedicato a 200 MHz, un chip audio a 64 canali e un disco ottico proprietario da 1,5 GB.

Anche se queste macchine non sono altrettanto potenti quanto i personal computer prodotti nello stesso periodo, esse non sono molto distanti da quest'ultimi e sotto alcuni aspetti sono addirittura più evolute (la CPU a 128 bit della PlayStation 2, per esempio, è molto più avanzata delle CPU di qualsiasi PC, sebbene la velocità di clock sia molto più bassa). La differenza principale tra una console e un PC non risiede tanto nella CPU quanto nel fatto che le prime sono sistemi chiusi. Anche se di solito le console hanno interfacce USB e FireWire, gli utenti non possono espanderle attraverso schede aggiuntive. Inoltre, e forse questo è l'aspetto più importante, le console sono attentamente ottimizzate per un particolare tipo di applicazioni, come i videogiochi 3D fortemente interattivi e con audio stereo di alta qualità. Tutto il resto è secondario. Queste restrizioni hardware e software, la bassa velotità di clock, la limitatezza della memoria, l'assenza di monitor ad alta risoluzione e (generalmente) l'assenza di un hard disk, rendono possibile la costruzione e la vendita di queste macchine a un prezzo nettamente inferiore rispetto a quello dei personal computer. Nonostante queste restrizioni, nel mondo sono stati venduti milioni di console.

Le stesse società che producono le principali console da gioco, producono anche console portatili che possono essere agevolmente tenute in mano e funzionano mediante batterie. Queste macchine sono più simili ai sistemi integrati illustrati precedentemente che ai personal computer.

1.3.6 Personal computer

Continuando a salire i gradini della scala degli elaboratori arriviamo ai personal computer, cioè quelle macchine a cui perlopiù si pensa quando si sente pronunciare il termine "computer". Essi comprendono i modelli desktop e quelli portatili. Generalmente sono forniti di centinaia di megabyte di memoria, un hard disk che può contenere circa 100 GB di dati, un lettore CD-ROM/DVD, un modem, una scheda audio, un'interfaccia di rete, un monitor ad alta risoluzione e altre periferiche. Sono dotati di sistemi operativi sofisticati, hanno molte possibilità di

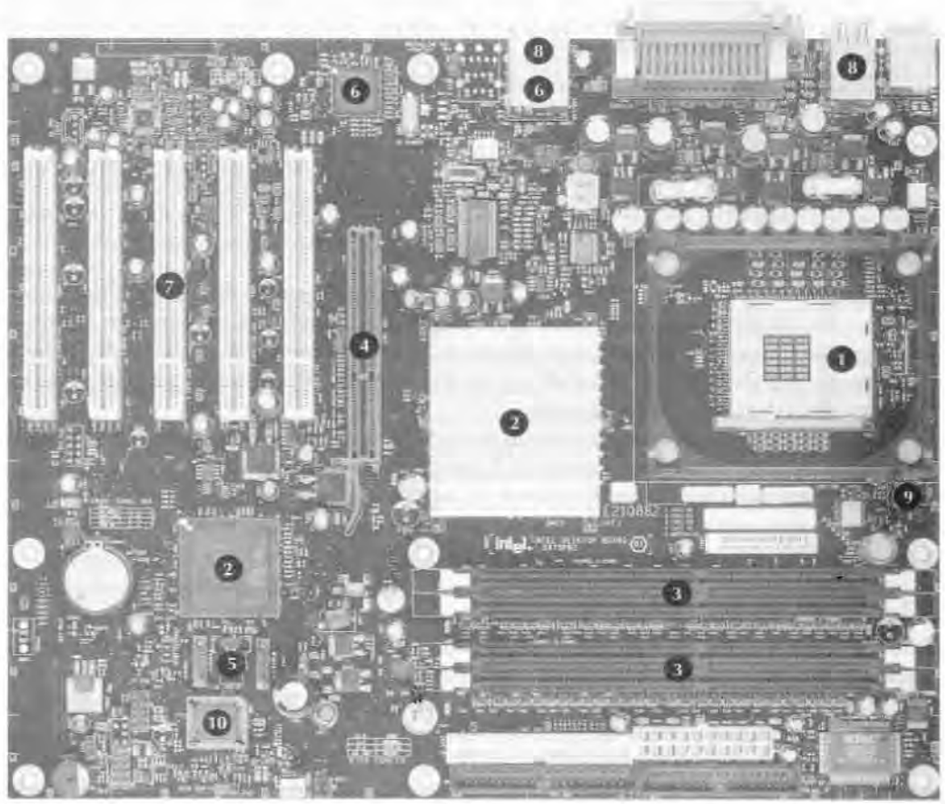


Figura 1.10 Nel cuore di ogni personal computer c'è una scheda di circuiti stampati. Questa immagine è una fotografia della scheda Intel D875PBZ. La fotografia è di proprietà di Intel Coorporation (copyright 2003) e pubblicata con autorizzazione.

1. Socket del Pentium 4
2. Chipset 875P
3. Socket per la memoria
4. Connettore AGP
5. Interfaccia per il disco
6. Ethernet Gigabit
7. Cinque slot PCI
8. Porte USB 2.0
9. Sistema di raffreddamento
10. BIOS

espansione e dispongono di un'ampia gamma di software disponibile. Alcuni riservano il termine "PC" alle macchine basate su CPU Intel, mentre usano il termine "workstation" per indicare quelle che usano un chip RISC ad alte prestazioni, come il Sun UltraSPARC. Concettualmente però tra i due vi è solamente una piccola differenza.

Il cuore di ogni personal computer è costituito da una scheda di circuiti stampati che contiene la CPU, la memoria, vari dispositivi di I/O (come un chip audio e talvolta un modem), oltre alle interfacce per tastiera, mouse, disco, rete e ad alcuni slot di espansione. La Figura 1.10 mostra un'immagine di una di queste schede.

I computer portatili sono fondamentalmente dei PC realizzati in dimensioni ridotte. Usano le stesse componenti hardware, seppur costruite in dimensioni inferiori, e possono eseguire lo stesso software dei PC da scrivania.

Un altro tipo di macchina strettamente collegato al personal computer è il PDA. Anche se le loro dimensioni sono ancora minori di quelle dei computer portatili, dispongono tuttavia di CPU, memoria, tastiera, schermo e di molte altre caratteristiche tipiche di un PC in miniatura. Dato che con ogni probabilità la maggior parte dei lettori ha familiarità con i personal computer, non vi è bisogno di ulteriore materiale introduttivo.

1.3.7 Server

Spesso vengono impiegate versioni potenziate dei personal computer o delle workstation come server di rete, sia per reti locali (di solito all'interno di un'azienda) sia per Internet. Esistono configurazioni mono o multiprocessore, dotate di gigabyte di memoria, centinaia di gigabyte di spazio su hard disk e connessioni di rete ad alta velocità. Alcuni server sono in grado di gestire migliaia di transazioni al secondo.

Dal punto di vista dell'architettura un server monoprocessore non è però molto diverso da un personal computer monoprocessore. È semplicemente più veloce e più grande, ha una maggiore memoria di massa e generalmente ha una connessione alla rete più veloce. I server eseguono gli stessi sistemi operativi dei personal computer, di solito uno dei dialetti di UNIX oppure di Windows.

1.3.8 Cluster di workstation

Dato che il rapporto prezzo/prestazioni delle workstation continua progressivamente a migliorare, negli ultimi anni i progettisti di sistemi hanno cominciato a connettere fra loro un gran numero di queste macchine per formare i cosiddetti **COW** (*Clusters of Workstations*, "gruppi di workstation"), detti anche *cluster*. Essi consistono di normali personal computer o workstation connessi fra loro mediante reti la cui velocità è nell'ordine dei gigabyte al secondo. Queste macchine eseguono software speciale che permette loro di lavorare in modo congiunto su uno stesso problema, spesso di tipo scientifico o ingegneristico. Generalmente le macchine utilizzate vengono chiamate COTS (*Commodity Off The Shelf*, "prodotti pronti per l'uso") cioè elaboratori che chiunque può acquistare in un normale negozio di computer. La principale aggiunta che viene fatta è una connessione di rete ad alta velocità, anche se talvolta si utilizzano normali schede di rete commerciali. I cluster sono facilmente scalabili; possono essere composti da poche macchine fino a migliaia di elementi e spesso il fattore che ne limita il numero è semplicemente il costo da affrontare. Grazie alla convenienza economica delle loro componenti, oggi anche un singolo dipartimento può permettersi di possedere tali macchine.

I cluster possono anche essere utilizzati come server Web per Internet. Quando si devono gestire migliaia di richieste al secondo per le pagine di un sito web spesso la soluzione più economica è un cluster con centinaia, se non migliaia, di server. Le richieste entranti vengono distribuite sui server per permettere loro di elaborarle in parallelo. Quando un cluster viene utilizzato in questo modo, spesso lo si chiama *server farm*, "fattoria di server".

1.3.9 Mainframe

In cima alla scala di computer troviamo i mainframe, computer grandi come una stanza che rievocano gli elaboratori degli anni '60. In molti casi queste macchine sono i diretti discendenti dei mainframe IBM 360. La maggior parte di loro ha una velocità che non è molto più elevata rispetto a quella dei server più potenti, ma tutti i mainframe hanno capacità di I/O maggiori e sono spesso equipaggiati con un vasto numero di dischi, per la memorizzazione di migliaia di gigabyte di dati. Pur essendo costosi vengono ancora utilizzati per via dell'enorme investimento che rappresentano in termini di software, dati, procedure operative e personale specializzato. Molte società ritengono che sia più conveniente pagare una volta ogni tanto alcuni milioni di euro per comprarne uno nuovo, piuttosto che prendere in considerazione lo sforzo necessario a riprogrammare tutte le loro applicazioni per macchine più piccole.

È questa classe di computer che ha portato all'ormai famoso bug del millennio, provocato dal fatto che i programmatori COBOL degli anni '60 e '70 usavano (per risparmiare memoria) solo due cifre decimali per rappresentare gli anni. Non avrebbero mai pensato che il loro software sarebbe durato trenta o quarant'anni. Grazie all'enorme lavoro profuso per risolvere il problema, il disastro previsto non si è verificato; ciononostante molte società sono ricadute nello stesso errore aggiungendo agli anni altre due cifre decimali. L'autore prevede quindi che la fine della civiltà così come l'abbiamo conosciuta avverrà il 31 dicembre 9999, quando l'equivalente di 8000 anni di vecchio codice COBOL smetterà simultaneamente di funzionare.

Negli ultimi anni, Internet ha dato nuova linfa ai mainframe, fino ad allora usati principalmente per l'esecuzione di software vecchio di 30 anni. Queste macchine hanno trovato una nuova nicchia come potenti server Internet gestendo per esempio un massiccio numero di transazioni di commercio elettronico al secondo, in particolare laddove sono richieste enormi basi di dati. Sebbene il libro sia dedicato ai PC, ai server e ai microcontrollori, nel Capitolo 5 analizzeremo più nel dettaglio anche i mainframe.

Fino a poco tempo fa esisteva anche un'ulteriore categoria di computer, ancora più potente dei mainframe: i **supercomputer**. Avevano CPU incredibilmente veloci, molti gigabyte di memoria principale e dischi e reti ad alta velocità. Queste macchine venivano utilizzate per imponenti calcoli scientifici e ingegneristici come la simulazione della collisione fra galassie, la sintesi di nuove medicine o la simulazione del flusso di aria attorno alle ali degli aerei. Negli ultimi anni i COW hanno però offerto la stessa potenza computazionale a un costo decisamente inferiore, e i veri supercomputer sono diventati una razza in via d'estinzione.

1.4 Esempi di famiglie di computer

In questo libro focalizzeremo l'attenzione su tre tipi di macchine: i personal computer, i server e i sistemi integrati. L'interesse verso i primi è motivato dal fatto che ogni lettore ne ha senza dubbio utilizzato uno, mentre quello verso i server è dovuto al fatto che eseguono tutti i servizi di Internet. Per quanto riguarda infine i computer integrati essi sono invisibili agli utenti, ma controllano macchine, televisori, forni a microonde, lavatrici e praticamente qualsiasi altro dispositivo che costi più di 50 euro.

In questo paragrafo daremo una breve introduzione dei tre computer che saranno usati come esempi nel resto del libro, uno per ognuna delle tre categorie sopracitate. Sono il Pentium 4, l'UltraSPARC III e l'8051.

1.4.1 Introduzione al Pentium 4

Nel 1968 Robert Noyce, inventore dell'integrazione dei circuiti su silicio, Gordon Moore, famoso per la legge omonima, e Arthur Rock, un imprenditore di San Francisco, fondarono la Intel Corporation per produrre chip di memoria. Durante il primo anno di attività il fatturato Intel fu di soli 3000 dollari, ma da allora gli introiti sono decollati.

Nei tardi anni '60 i calcolatori erano grandi macchine elettromeccaniche della grandezza di una moderna stampante laser e del peso di 20 Kg. Nel settembre del 1969 una società giapponese, la Busicom, si rivolse a Intel per richiedere la produzione di 12 chip appositamente progettati per un nuovo calcolatore elettronico. Ted Hoff, l'ingegnere di Intel cui fu affidato l'incarico, analizzò il progetto e si rese conto che, per realizzare le stesse operazioni, avrebbe potuto mettere su un singolo chip una CPU a 4 bit di uso generale e che questa sarebbe stata allo stesso tempo più semplice ed economica. Così nel 1970 nacque il processore 4004, la prima CPU su un chip costituita da 2300 transistor (Faggin et al., 1996).

Occorre notare che né Intel né Busicom si resero subito conto di che cosa avessero appena realizzato. Quando Intel decise che valeva la pena provare a utilizzare il 4004 in altri progetti, propose a Busicom di ricomprare tutti i diritti, restituendole i 60.000 dollari che la società giapponese aveva pagato per lo sviluppo della CPU. L'offerta fu subito accettata e da quel momento Intel poté cominciare a lavorare su una versione a 8 bit del processore, l'8008, introdotto nel 1972. La famiglia Intel, iniziata con il 4004 e l'8008 è illustrata nella Figura 1.11.

Intel non si aspettava una grande richiesta per l'8008 e quindi impostò una linea di produzione su un basso volume. Fra lo stupore generale, il chip riscosse un enorme interesse, tale da spingere Intel a progettare una nuova CPU che aggirasse il principale problema dell'8088, cioè il limite di 16 KB di memoria (dovuto al numero di contatti esterni del chip). Il progetto terminò nel 1974 con la nascita dell'8080, una piccola CPU di uso generale. Esattamente come avvenne per il PDP-8, questo prodotto conquistò di colpo il mercato e divenne improvvisamente un articolo con un mercato di massa. La differenza è che, invece di venderne migliaia come fece DEC, Intel riuscì a venderne milioni.

Nel 1978 comparve l'8086, un'autentica CPU a 16 bit su un solo chip. L'8086 fu progettato per essere simile all'8080, pur senza la completa compatibilità. L'8086 fu seguito dall'8088, che aveva la sua stessa architettura e poteva eseguire gli stessi programmi, ma era dotato di un bus a 8 invece che a 16 bit, rendendolo più lento e allo stesso tempo più economico dell'8086. Quando IBM scelse l'8088 come CPU per il primo PC IBM, questo processore divenne rapidamente lo standard nel mercato dei personal computer.

Chip	Data	MHz	N. di transistor	Memoria	Descrizione
4004	4/1971	0,108	2300	640	Primo microprocessore su un solo chip
8008	4/1972	0,108	3500	16 KB	Primo microprocessore a 8 bit
8080	4/1974	2	6000	64 KB	Prima CPU di uso generale su un solo chip
8086	6/1978	5-10	29.000	1 MB	Prima CPU a 16 bit su un solo chip
8088	6/1979	5-8	29.000	1 MB	Usato nel PC IBM
80286	2/1982	8-12	134.000	16 MB	Introduzione della modalità protetta
80386	10/1985	16-33	275.000	4 GB	Prima CPU a 32 bit
80486	4/1989	25-100	1,2 M	4 GB	Memoria cache da 8 KB integrata
Pentium	3/1993	60-233	3,1 M	4 GB	Due pipeline; istruzioni MMX nei modelli successivi
Pentium Pro	3/1995	150-200	5,5 M	4 GB	Cache integrata a due livelli
Pentium II	5/1997	233-450	7,5 M	4 GB	Pentium Pro con istruzioni MMX
Pentium III	2/1999	650-1400	9,5 M	4 GB	Istruzioni SSE per la grafica 3D
Pentium 4	11/2000	1300-3800	42 M	4 GB	Hyperthreading; ulteriori istruzioni SSE

Figura 1.11 La famiglia di CPU Intel. Le velocità del clock sono misurate in MHz (megahertz) dove 1 Mhz corrisponde a 1 milione di cicli/s.

Né l'8088 né l'8086 potevano indirizzare più di 1 MB di memoria, il che nei primi anni '80 divenne via via un serio problema, al punto che Intel decise di progettare l'80286, una versione più potente, ma ancora compatibile con l'8086. L'insieme base d'istruzioni era essenzialmente lo stesso dell'8086 e dell'8088, ma l'organizzazione della memoria era molto differente e piuttosto complessa, a causa dei requisiti di compatibilità con gli altri processori. L'80286 fu utilizzato nel PC IBM/AT e nei modelli PS/2 di gamma media. Come l'8088 fu anch'esso un grande successo, soprattutto perché era in genere considerato come un 8088 più veloce.

Il successivo passo fu logicamente l'80386, una vera CPU a 32 bit su un chip che vide la luce nel 1985. Come il precedente, questo processore era una versione più o meno compatibile con tutto ciò che risaliva all'8080. Essendo retrocompatibile fu una benedizione per chi aveva necessità di eseguire vecchio software, ma un fastidio per chi avrebbe preferito un'architettura moderna, semplice, ben progettata e non appesantita dagli errori e dalla tecnologia degli anni precedenti.

Quattro anni dopo apparve l'80486. Era in sostanza una versione più veloce dell'80386 con in più un'unità in virgola mobile e una memoria cache di 8 KB sul chip. La **memoria cache** era usata per conservare, all'interno o vicino alla CPU, le parole di memoria utilizzate con più frequenza, di modo da evitare l'accesso (lento) alla memoria centrale. L'80486 aveva anche un supporto multiprocessore predefinito per permettere ai produttori di costruire sistemi composti da più CPU che condividessero una memoria comune.

A quel punto Intel scoprì nel modo peggiore (perdendo una causa legale per violazione di diritti) che i numeri (come 80486) non potevano essere registrati come marchio, e per

questo alla generazione successiva fu dato un nome: Pentium (dal termine greco "cinque", *πέντα*). Diversamente dall'80486, che aveva una sola pipeline interna, il Pentium ne aveva due, il che lo aiutò a essere due volte più veloce (le pipeline verranno illustrate in dettaglio nel Capitolo 2).

In seguito, durante la sua corsa produttiva, Intel aggiunse l'insieme d'istruzioni speciali **MMX** (*MultiMedia eXtension*, "estensioni multimediali"). Queste istruzioni furono ideate per velocizzare i calcoli necessari per l'elaborazione di audio e video, rendendo così superflua la presenza di coprocessori multimediali.

Quando apparve la generazione successiva, la gente che sperava di vedere il Sexium (*sex* significa sei in Latino) rimase delusa. Ormai il nome Pentium era talmente conosciuto che gli addetti al marketing lo vollero mantenere e il nuovo chip prese il nome di Pentium Pro. Nonostante il piccolo cambiamento nel nome, questo processore rappresentò una maggiore discontinuità rispetto al passato. Invece di avere due o più pipeline il Pentium Pro aveva un'organizzazione interna molto diversa e poteva eseguire fino a cinque istruzioni allo stesso tempo.

Un'altra innovazione introdotta dal Pentium Pro fu la memoria cache a due livelli. Nel chip stesso del processore vi erano 8 KB di memoria veloce contenenti le istruzioni usate più di frequente e altri 8 KB dedicati in modo analogo ai dati. All'interno della stesso alloggiamento del Pentium Pro (ma non nel chip stesso) vi era una seconda memoria cache di 256 KB.

Anche se il Pentium Pro era dotato di una cache di grandi dimensioni, mancavano le istruzioni MMX (dato che Intel non riuscì a produrre un così grande chip a un costo accettabile). Quando i progressi tecnologici permisero di avere contemporaneamente sullo stesso chip sia le istruzioni MMX sia la cache, il risultato fu la nascita del Pentium II. Successivamente, per migliorare le prestazioni con la grafica 3D, furono aggiunte ulteriori istruzioni multimediali, chiamate **SSE** (*Streaming SIMD Extensions*, "estensioni per stream SIMD") (Raman et al., 2000). Al nuovo chip fu dato il nome di Pentium III, anche se internamente era in sostanza un Pentium II.

Il Pentium successivo fu basato su un'architettura interna differente. Per celebrare l'evento Intel passò dalla numerazione in numeri romani a quella in numeri arabi e lo chiamò Pentium 4. Come al solito il Pentium 4 era più veloce di tutti i suoi predecessori; la versione a 3,06 GHz introdusse in più una nuova e interessante caratteristica: l'*hyperthreading*. Questa potenzialità permetteva ai programmi di dividere il loro lavoro in due flussi di controllo che il Pentium 4 poteva eseguire in parallelo, accelerandone l'esecuzione. Inoltre vennero aggiunte altre istruzioni SSE per velocizzare ulteriormente l'elaborazione audio e video. La Figura 1.12 mostra una fotografia del chip del Pentium 4; le sue dimensioni reali (circa 16×13,5 mm) ne fanno un chip particolarmente grande.

Intel, oltre alla linea principale di CPU per computer desktop, ha prodotto varianti di alcuni processori Pentium dedicate a mercati specifici. Nel 1998 ha introdotto una nuova linea di prodotti chiamata **Celeron**, in sostanza una versione più economica e con prestazioni ridotte del Pentium II, pensata per i PC di fascia bassa. Dato che l'architettura del Celeron è la stessa del Pentium II, non verrà trattata nel corso del libro. Nel giugno del 1998 Intel ha introdotto anche una versione speciale del Pentium II dedicata al mercato professionale. Questo processore, chiamato **Xeon**, aveva una cache più grande, un bus più veloce e un miglior supporto multiprocessore; a parte questi miglioramenti si trattava di un normale Pentium II, quindi neanch'esso verrà trattato specificatamente. Anche del Pentium III è stata prodotta una versione Xeon.

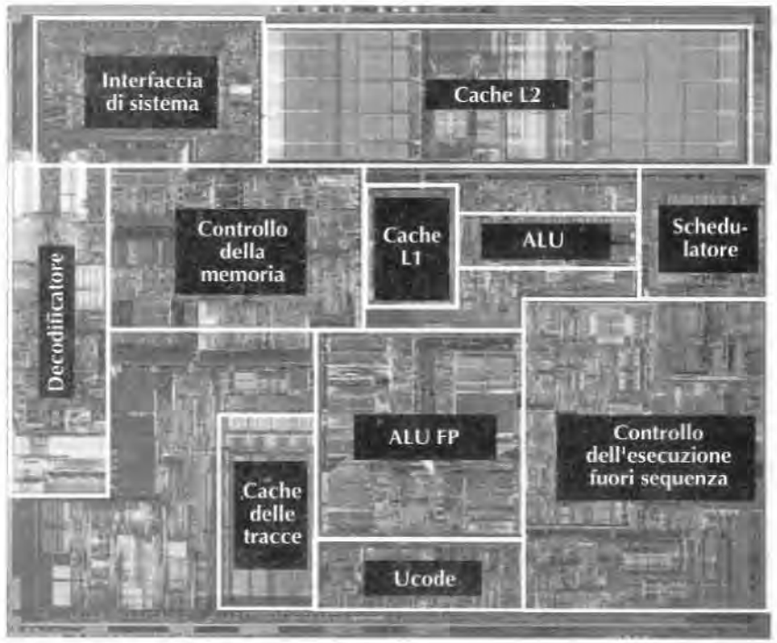


Figura 1.12 Il chip Pentium 4. La fotografia è di proprietà di Intel Coorporation (copyright 2003) e pubblicata con autorizzazione.

Nel novembre del 2000 Intel ha rilasciato il Pentium 4, un processore capace di eseguire gli stessi programmi del Pentium III e di Xeon, ma con una progettazione interna completamente nuova. La versione a 3,06 GHz ha introdotto l'hyperthreading, un argomento che affronteremo nel Capitolo 8.

Nel 2003 Intel ha messo in commercio il Pentium M (M per Mobile), progettato per i computer portatili. Questo chip faceva parte dell'architettura Centrino, i cui obiettivi erano la riduzione del consumo energetico per prolungare la durata delle batterie, la produzione di computer più piccoli e leggeri e la predisposizione alla connessione di rete senza fili secondo lo standard IEEE 802.11 (WiFi). Intel intende introdurre nel prossimo futuro nuovi chip dedicati ad altri campi specifici come, per esempio, gli apparecchi per l'intrattenimento e i portatili con connessione IEEE 802.16 (WiMax).

Tutti i processori Intel sono retrocompatibili con i loro predecessori fino all'8086. In altre parole un Pentium 4 può eseguire programmi dell'8086 senza alcuna modifica. Per Intel questa compatibilità è sempre stata un requisito di progettazione, al fine di non far perdere agli utenti i loro investimenti in software. Il Pentium 4 è però tre ordini di grandezza più complesso dell'8086 ed è quindi naturale che possa fare molte cose in più. L'aggiunta graduale di tutte queste estensioni ha portato a un'architettura che non è così elegante come quella che si otterrebbe dando a qualcuno 42 milioni di transistor e il compito di progettarla ex novo.

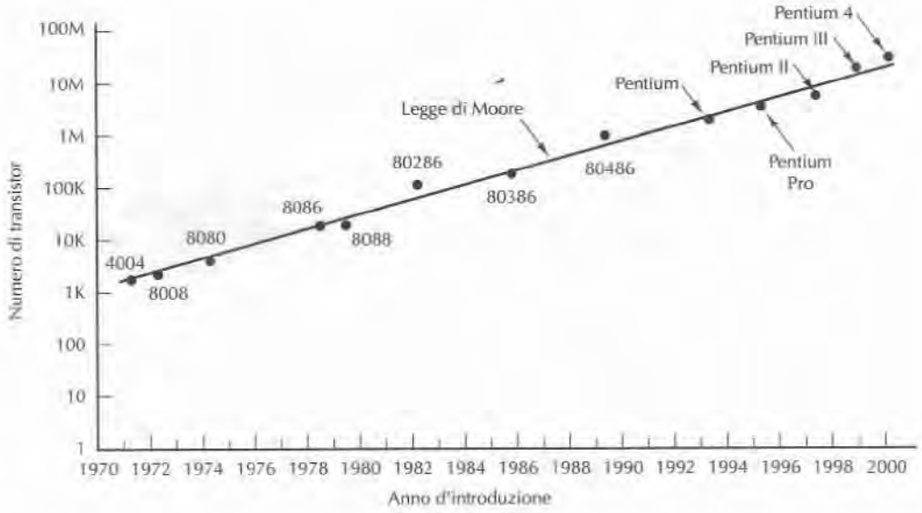


Figura 1.13 La legge di Moore per i chip della CPU (Intel).

È interessante notare che la legge di Moore, sebbene sia stata a lungo associata al numero di bit di memoria, risulta valida anche per i chip della CPU. Il grafico nella Figura 1.13 rappresenta in scala semi-logaritmica il numero di transistor di un chip rispetto alla sua data di apparizione e mostra che la legge di Moore viene rispettata anche in base a questi dati.

Anche se con ogni probabilità la legge di Moore continuerà a essere valida per alcuni anni a venire, cominciano a profilarsi delle nuvole all'orizzonte per via dei problemi legati alla dissipazione del calore. L'uso di transistor più piccoli permette di raggiungere frequenze più alte, ma allo stesso tempo richiede una tensione più elevata. L'energia consumata e il calore dissipato sono proporzionali al quadrato della tensione, ragion per cui raggiungere velocità più elevate significa avere più calore da smaltire. Un processore Pentium 4 a 3,6 GHz consuma 115 W, il che significa che diventa caldo all'incirca quanto una lampadina da 100 W. Aumentare la frequenza di clock peggiora il problema.

Nel novembre del 2004 Intel ha dovuto rinunciare al Pentium 4 a 4 GHz a causa dei problemi dipendenti dalla dissipazione del calore. L'uso di ventole di grandi dimensioni può aiutare, ma il loro rumore infastidisce gli utenti e il raffreddamento ad acqua, usato nei mainframe di grandi dimensioni, non è un'opzione attualmente praticabile per i desktop (e ancor meno per quelli portatili). Per questo motivo la finora inarrestabile marcia delle velocità di clock potrebbe essere temporaneamente arrestata, almeno finché gli ingegneri di Intel non escogiteranno un modo per liberarsi efficientemente del calore generato. I futuri piani di Intel prevedono di collocare due CPU su un singolo chip e di utilizzare una grande cache condivisa. Per via della relazione tra tensione e frequenza di clock, due CPU sullo stesso chip consumano molta meno potenza di una sola CPU con velocità doppia. Ne consegue che in futuro il guadagno fornito dalla legge di Moore potrebbe essere via via sfruttato per inserire nel chip cache sempre più grandi, piuttosto che per raggiungere velocità di clock sempre più elevate (dato che la memoria non consuma molta potenza).

1.4.2 Introduzione all'UltraSPARC III

Negli anni '70 UNIX era diffuso nelle università ma, dato che non era usato nei personal computer, i suoi appassionati erano costretti a usare minicomputer a condivisione di tempo (spesso sovraccaricati) come i PDP-11 e i VAX. Nel 1981 Andy Bechtolsheim, uno studente tedesco laureatosi a Standford, stanco di doversi recare fino al centro computer per poter utilizzare UNIX, decise di risolvere il problema costruendo da sé una workstation UNIX mediante parti facilmente reperibili nei negozi. Chiamò questa macchina SUN-1 (*Stanford University Network*).

Bechtolsheim attrasse ben presto l'attenzione di Vinod Khosla, un ventisettenne indiano il cui ardente desiderio era andare in pensione, da milionario, alla giovane età di 30 anni. Khosla convinse Bechtolsheim a fondare una società per costruire e vendere le workstation Sun. Khosla assunse quindi un altro studente laureatosi a Standford, Scott McNealy, per guidare la produzione. Per scrivere il software venne incaricato invece Bill Joy, il principale sviluppatore di Berkley UNIX. Nel 1982 i quattro fondarono la Sun Microsystems.

Il primo prodotto di Sun, il computer Sun-1, era basato su una CPU Motorola 68020 e fu un istantaneo successo; allo stesso modo lo furono le macchine che lo seguirono, i computer Sun-2 e Sun-3, anch'essi equipaggiati da CPU Motorola. Diversamente dagli altri personal computer dell'epoca, queste macchine avevano una potenza decisamente maggiore (da qui il termine "workstation") ed erano progettate fin dall'inizio per poter funzionare in rete. Ciascuna workstation Sun era dotata di una connessione Ethernet e di software TCP/IP per la connessione ad ARPANET, il precursore di Internet.

Nel 1987 Sun, che ormai rivendeva ogni anno sistemi per un valore di mezzo miliardo di dollari, decise di progettare una propria CPU, basata su un nuovo rivoluzionario progetto in corso nel campus di Berkeley della University of California (il RISC II). Questa CPU, chiamata **SPARC** (*Scalable Processor ARChitecture*, "processore ad architettura scalabile"), costituì la base della workstation Sun-4. In breve tempo tutti i prodotti di Sun usarono la CPU SPARC.

Diversamente dalle altre società di computer Sun decise di non produrre da sola le CPU SPARC, ma diede a diversi produttori di semiconduttori la licenza di produrle, sperando che la concorrenza avrebbe fatto crescere le prestazioni e diminuire i prezzi. Questi produttori realizzarono una serie di diversi processori, basati su tecnologie differenti, funzionanti a diverse frequenze di clock e a vari prezzi. Alcuni di questi chip furono il MicroSPARC, l'HyperSPARC, il SuperSPARC e il TurboSPARC. Pur distinguendosi in qualche aspetto secondario, tutti erano compatibili a livello binario e potevano eseguire gli stessi programmi utente senza alcuna modifica.

Sun ha sempre desiderato che SPARC restasse un'architettura aperta, con molti fornitori di componenti e di sistemi, per poter creare un mercato che riuscisse a competere in un mondo dei personal computer già dominato dai sistemi basati su CPU Intel. Al fine di conquistare la fiducia di quelle società che, pur essendo interessate a SPARC, erano restie a investire in un prodotto controllato da un concorrente, SPARC creò SPARC International, un consorzio di industrie per gestire lo sviluppo delle nuove versioni dell'architettura. Per questa ragione è importante distinguere tra l'architettura SPARC, che

è una specifica dell'insieme d'istruzioni e delle funzionalità visibili al programmatore, e una sua particolare implementazione. In questo libro studieremo sia la generica architettura SPARC sia, nel corso dei Capitoli 3 e 4 dedicati ai chip delle CPU, uno specifico processore SPARC usato nelle workstation Sun.

La prima macchina SPARC era interamente a 32 bit e funzionava a 36 MHz. La CPU, chiamata **IU** (*Integer Unit*, "unità di elaborazione intera"), era semplice e ben progettata, con solamente 55 istruzioni basate su 3 formati principali. Un'unità in virgola mobile aggiungeva 14 ulteriori istruzioni. Questa storia può essere contrapposta a quella della linea di processori Intel, che inizialmente furono a 8 e 16 bit (8088, 8086, 80286) e infine, con l'80386, diventarono processori a 32 bit.

Nel 1995 lo sviluppo della Version 9 dell'architettura SPARC rappresentò la prima rottura di SPARC con il passato. Si trattava di un'architettura interamente a 64 bit, con indirizzi e registri a 64 bit. La prima workstation Sun a implementare la V9 (Version 9) fu l'**UltraSPARC I** che apparve nel 1995 (Tremblay e O'Connor, 1996). Nonostante fosse una macchina a 64 bit era completamente compatibile a livello binario con le precedenti macchine SPARC a 32 bit.

L'UltraSPARC era stata pensata per superare nuovi confini. Mentre le macchine precedenti erano progettate per gestire dati alfanumerici ed eseguire programmi per il trattamento testi e fogli di calcolo, l'UltraSPARC era invece progettata fin dall'inizio per gestire immagini, audio, video e dati multimediali. Al di là dell'architettura a 64 bit, una delle altre innovazioni fu rappresentata da 23 nuove istruzioni, comprese alcune per compattare e scompattare pixel da parole di 64 bit, per ruotare e ridimensionare immagini, per spostare blocchi e per comprimere e decomprimere video in tempo reale. Queste istruzioni, chiamate **VIS** (*Visual Instruction Set*, "insieme d'istruzioni multimediali"), erano finalizzate a fornire funzionalità multimediali generiche, analogamente alle istruzioni MMX di Intel.

L'UltraSPARC era destinato ad applicazioni professionali, come i grandi server Web multiprocessori costituiti da dozzine di CPU e memorie fisiche fino a 8 Tbyte = 10^{12} byte. Tuttavia esistevano versioni più piccole utilizzabili per i computer portatili.

I successori di questo computer furono gli UltraSPARC II, III e IV. La differenza principale fra questi modelli era la frequenza del clock, anche se a ogni nuova versione vennero comunque aggiunte caratteristiche innovative. Nella maggior parte del libro, quando tratteremo l'architettura SPARC, utilizzeremo principalmente come esempio la CPU V9 UltraSPARC III a 64 bit. L'UltraSPARC IV è essenzialmente un sistema biprocessore in cui due UltraSPARC III sono collocate sullo stesso chip e condividono la stessa memoria. Tratteremo questo processore nel Capitolo 8 quando affronteremo i sistemi multiprocessore.

1.4.3 Introduzione all'8051

Il terzo esempio che verrà utilizzato è molto differente rispetto ai precedenti (usati nei personal computer e nei server). Si tratta dell'8051, un processore che viene impiegato nei sistemi integrati. La storia dell'8051 cominciò nel 1976, quando il processore 8080 a 8 bit era già sul mercato da circa due anni. I costruttori di apparecchi elettronici stavano cominciando a integrare l'8080 nei loro dispositivi, ma per costruire un sistema completo avevano bisogno di un chip contenente la CPU 8080, di uno o più chip di memoria e di uno o più

chip per l'I/O. Il costo di almeno tre chip e delle loro interconnessioni era rilevante e perciò l'integrazione di computer rimase limitata a prodotti piuttosto grandi e costosi. Per ridurre i costi molti produttori chiesero a Intel di integrare l'intero computer (CPU, memoria e I/O) all'interno di un solo chip.

La risposta di Intel ai suoi clienti fu la produzione del chip 8748, un microcontrollore da 17.000 transistor contenente una CPU simile all'8080, 1 KB di memoria di sola lettura, 64 byte di memoria di lettura e scrittura per le variabili, un timer a 8 bit e 27 linee di I/O per gestire interruttori, pulsanti e luci. Pur essendo molto essenziale, il chip fu un successo commerciale che portò Intel a rilasciare nel 1980 l'8051. Questo nuovo chip conteneva 60.000 transistor, una CPU molto più veloce, 4 KB di memoria di sola lettura, 64 byte di memoria di lettura e scrittura, 32 linee di I/O, una porta seriale e due timer a 16 bit. Ben presto fu seguito da altri membri di quella che Intel chiamò la **famiglia MCS-51** (mostrata nella Figura 1.14).

Tutti questi chip utilizzavano una memoria di sola lettura per il programma oltre a una piccola quantità di memoria RAM di lettura e scrittura, per i dati. Nel caso dell'8031 e dell'8032 la memoria per il programma era esterna, permettendo, se necessario, di usare più di 8 KB. Nel Capitolo 3 studieremo le **ROM** (*Read Only Memory*, "memoria di sola lettura") e le **EPROM** (*Erasable Programmable ROM*, "ROM cancellabile e programmabile").

Per il momento è sufficiente sapere che l'8051 e l'8052 sono microcontrollori su singolo chip utilizzati in prodotti attualmente in vendita. Ciascun lotto di microcontrollori è prodotto appositamente per il cliente (per esempio un costruttore di elettrodomestici) e contiene il programma fornito da quest'ultimo.

Per programmare il cliente necessita tuttavia di un sistema di sviluppo, ed è qui che entrano in gioco l'8751 e l'8752. Sono molto più costosi dell'8051 e dell'8052, ma possono essere programmati direttamente dal cliente per permettergli di testare il software. Se viene individuato un bug è possibile cancellare la memoria dell'8751 o dell'8752 esponendo il chip a una luce ultravioletta, così da poter successivamente memorizzare al loro interno un nuovo programma. Quando lo sviluppo del software è terminato, può essere consegnato al produttore del chip, che può così realizzare versioni dell'8051 o dell'8052 adattate al codice.

Dal punto di vista dell'architettura, dell'interfaccia e della programmazione tutti i membri della famiglia MCS-51 sono molto simili. Per semplicità ci riferiremo principalmente all'8051, sottolineando le differenze con gli altri chip solo quando sarà necessario.

Chip	Memoria per il programma	Tipo di memoria	RAM	Numero di timer	Numero di interrupt
8031	0 KB		128	2	5
8051	4 KB	ROM	128	2	5
8751	8 KB	EPROM	128	2	5
8032	0 KB		256	3	6
8052	8 KB	ROM	256	3	6
8751	8 KB	EPROM	256	3	6

Figura 1.14 Membri della famiglia MCS-51.

Ad alcuni sembrerà strano utilizzare come esempio un chip a 8 bit che ha più di 20 anni di età, ma vi sono alcune buone ragioni per farlo. Il numero di microcontrollori venduti ogni anno aumenta rapidamente ed è oggi di circa 8 miliardi; un valore che è vari ordini di grandezza superiore rispetto al numero di Pentium venduti annualmente. Soltanto nel 2001 il numero di microcontrollori a 8 bit ha superato il volume di vendita dei microcontrollori a 4 bit. Attualmente i microcontrollori a 8 bit vendono più di tutti gli altri messi insieme e la famiglia MCS-51 è la famiglia più popolare a 8 bit. Data la crescente importanza dei sistemi integrati, chiunque studi le architetture dei computer dovrebbe avere familiarità con i chip usati al loro interno e l'8051 è uno dei più diffusi.

Vi sono varie ragioni alla base del successo dell'8051: il prezzo è la prima e principale. A seconda del numero di unità ordinate, un 8051 può essere ottenuto per 10 o 15 centesimi al chip, forse meno nel caso di grandi volumi. Al contrario, un microcontrollore a 32 bit costa generalmente 30 volte di più, mentre uno a 16 bit ha un prezzo più o meno intermedio. Nel caso di prodotti venduti a meno di 50 euro su mercati competitivi, tagliare un paio di euro dai costi di produzione può tradursi in una differenza significativa nel prezzo al dettaglio e nelle vendite. La ragione principale per cui l'8051 è talmente diffuso è semplicemente che è molto economico.

La seconda ragione è che oltre mezza dozzina di società producono i chip 8051 sotto la licenza di Intel. I loro prodotti impiegano diverse tecnologie di produzione e le loro velocità variano lungo un ampio spettro, dagli originari 12 MHz fino a 100 MHz. Questa competizione tiene bassi i prezzi e fa sì che i numerosi clienti siano molto più soddisfatti, dato che non devono dipendere da un unico fornitore.

La terza ragione è che l'8051 è stato in commercio per così tanto tempo che vi è una vasta quantità di software disponibile, tra cui assemblatori, compilatori per C e altri linguaggi, librerie di tutti i tipi, debugger, simulatori, software per effettuare test e molto altro. Esistono sul mercato anche molti completi sistemi di sviluppo, che accelerano la creazione di hardware e software integrati. Infine è facile trovare personale qualificato, dato che un gran numero di programmatori e ingegneri elettronici ha familiarità con l'8051.

La popolarità si autoalimenta. Grazie al suo impiego molto diffuso i ricercatori interessati ai sistemi integrati spesso scelgono l'8051 come loro principale oggetto di studio, per esempio per testare nuove tecnologie efficienti sotto il profilo energetico (Martin et al., 2003) o per studiare la tolleranza ai guasti (Lima et al., 2002).

In Internet esiste una gran quantità d'informazioni sull'8051 e un buon punto di partenza può essere il sito www.8051.com. Inoltre ancora oggi vi sono persone che continuano a scrivere nuovi libri su questo processore (Ayala, 2004; Calcutt et al., 2004; MacKenzie et al., 2005; Mazidi et al., 2005).

1.5 Unità metriche

Per evitare ogni tipo di confusione, vale la pena precisare che in questo libro, come di solito avviene nell'informatica, si usano le unità metriche e non le tradizionali unità di misura inglesi.

Nella Figura 1.15 sono elencati i principali prefissi metrici. Di solito sono abbreviati utilizzando le loro lettere iniziali e si usano lettere maiuscole per i multipli, come KB, MB, etc., e minuscole per i sottomultipli. Quindi, per esempio, una linea di comunicazione a 1 Mbps trasmette 10^6 bit/s e un clock a 100 ps ha un tic ogni 10^{-10} secondi. Dato che sia *milli* sia *micro* iniziano con la lettera "m" si utilizza "m" per milli e "μ" (la lettera greca mu) per micro.

Occorre inoltre sottolineare che, nella pratica comune, queste unità hanno un significato leggermente diverso quando vengono utilizzate come misure della dimensione di memorie, dischi, file e database. In questi casi, dato che le dimensioni delle memorie sono sempre potenze di due, K assume il significato di 2^{10} cioè 1024 invece che di $10^3 = 1000$. Quindi una memoria da 1 KB contiene 1024 e non 1000 byte. Analogamente una memoria da 1 MB contiene $2^{20} = 1.048.576$ byte, una memoria da 1 GB ne contiene $2^{30} = 1.073.741.824$, e un database da 1 Tbyte contiene $2^{40} = 1.099.511.627.776$ byte. Tuttavia, una linea di comunicazione a 1 Kbps può trasmettere 1000 bit al secondo e una LAN a 10 Mbps funziona a 10.000.000 bit/s, dato che questi tassi non sono potenze di due. Sfortunatamente molti tendono a confondere questi due sistemi, specialmente per le dimensioni dei dischi. Per eliminare ogni ambiguità in questo libro useremo i simboli KB, MB, GB, e Tbyte per indicare rispettivamente 2^{10}, 2^{20}, 2^{30} e 2^{40} byte e i simboli Kbps, Mbps, Gbps e Tbps per indicare rispettivamente 10^3, 10^6, 10^9 e 10^{12} bit/s.

Esp.	Valore esplicito	Prefisso	Esp.	Valore esplicito	Prefisso
10^{-3}	0,001	milli	10^{3}	1.000	Kilo
10^{-6}	0,000001	micro	10^{6}	1.000.000	Mega
10^{-9}	0,000000001	nano	10^{9}	1.000.000.000	Giga
10^{-12}	0,000000000001	pico	10^{12}	1.000.000.000.000	Tera
10^{-15}	0,000000000000001	femto	10^{15}	1.000.000.000.000.000	Peta
10^{-18}	0,000000000000000001	atto	10^{18}	1.000.000.000.000.000.000	Exa
10^{-21}	0,000000000000000000001	zepto	10^{21}	1.000.000.000.000.000.000.000	Zetta
10^{-24}	0,000000000000000000000001	yocto	10^{24}	1.000.000.000.000.000.000.000.000	Yotta

Figura 1.15 Principali prefissi metrici.

1.6 Organizzazione del libro

Questo libro tratta dei computer multilivello (che comprendono praticamente tutti i computer moderni) e del modo in cui sono organizzati. Verranno esaminati in notevole dettaglio quattro livelli: il livello logico digitale, il livello di microarchitettura, il livello ISA e il livello macchina del sistema operativo. Alcuni degli aspetti che esamineremo saranno la progettazione complessiva del livello (e il motivo per cui è stato progettato in quel modo), i tipi d'istruzioni e dati disponibili, l'organizzazione e l'indirizzamento della memoria e il metodo tramite il quale il livello è stato implementato. Lo studio di questi argomenti, e altri simili, è chiamato organizzazione, o architettura, del computer.

Ci preoccuperemo principalmente dei concetti piuttosto che dei dettagli e degli aspetti matematici. Per questa ragione alcuni degli esempi saranno fortemente semplificati, al fine di porre l'enfasi sulle idee centrali.

Nel corso del libro useremo il Pentium 4, l'UltraSPARC III e l'8051 come esempi pratici, in modo da far capire come i principi presentati nel testo possono essere, e sono, messi in pratica nella realtà. Questi tre esempi sono stati scelti in base a molteplici ragioni. In primo luogo sono tutti ampiamente utilizzati e con ogni probabilità il lettore può avere accesso ad almeno uno di loro. In secondo luogo, ciascuno ha una sua propria e determinata architettura, il che pone le basi per effettuare confronti e incoraggia un approccio basato su domande come: "Quali sono le alternative?". Spesso i libri che trattano di una sola macchina lasciano al lettore la sensazione che gli sia stata svelata "l'unica e corretta progettazione di un computer"; ciò è assurdo alla luce dei molti compromessi e delle decisioni arbitrarie che un progettista è obbligato a compiere. Si incoraggia il lettore a studiare questi computer, ed eventualmente altri, con un occhio critico; lo si spinge a cercare di capire il motivo per cui le cose sono in un determinato modo, oltre che a chiedersi come sarebbero potute essere diversamente, invece di accettarle passivamente.

Occorre chiarire fin dall'inizio che questo libro non spiega come programmare il Pentium 4, l'UltraSPARC III o l'8051. Quando sarà necessario, queste macchine saranno usate a scopo dimostrativo, senza alcuna pretesa di essere completi. I lettori interessati a un'introduzione esauriente a una di queste macchine dovrebbero consultare le pubblicazioni specifiche.

Il Capitolo 2 è un'introduzione ai componenti base dei computer: processori, memorie e dispositivi di input/output. Si intende dare una panoramica dell'architettura di un sistema oltre a un'introduzione ai capitoli successivi.

I Capitoli da 3 a 6 trattano individualmente i livelli mostrati nella Figura 1.2. La trattazione sarà effettuata dal basso verso l'alto, dato che le macchine sono state tradizionalmente progettate in questo modo.

La progettazione di un generico livello è in gran parte determinata dalle proprietà del livello sottostante e per questo è difficile capire un livello a meno di non avere prima compreso a fondo i precedenti. Inoltre sembra più istruttivo procedere dai livelli più semplici fino a quelli più complessi, e non viceversa.

Il Capitolo 3 riguarda il livello logico digitale, il vero e proprio hardware della macchina. Si spiega che cosa sono le porte logiche e come possono essere combinate per formare circuiti funzionali. Si introduce anche l'algebra booleana, uno strumento per l'analisi dei circuiti digitali. Vengono spiegati inoltre i bus del computer e in particolare il popolare bus PCI. Nel capitolo vengono presentati numerosi esempi tratti dal mondo reale, tra cui i tre casi di studio sopracitati.

Il Capitolo 4 introduce l'architettura e il controllo del livello di microarchitettura. Ricorrendo anche a vari esempi ci si concentrerà sulla funzione principale di questo livello, che consiste nell'interpretare le istruzioni del livello 2 per quello superiore. Il capitolo contiene anche una trattazione del livello di microarchitettura di alcune macchine reali.

Il Capitolo 5 esamina il livello ISA, quello che molti rivenditori di computer definiscono come il linguaggio macchina. In questo capitolo analizzeremo in dettaglio le tre macchine d'esempio.

Il Capitolo 6 si occupa delle istruzioni, dell'organizzazione della memoria e dei meccanismi di controllo presenti nel livello macchina del sistema operativo. Gli esempi utilizzati sono Windows XP (diffuso sui sistemi server di fascia alta basati su Pentium 4) e UNIX, usato sull'UltraSPARC III.

Il Capitolo 7 riguarda il livello del linguaggio assemblativo e tratta sia del linguaggio sia del processo di assemblaggio. Viene inoltre introdotto l'argomento del collegamento (*linking*).

Il Capitolo 8 tratta dei computer paralleli, un argomento la cui importanza al giorno d'oggi è sempre più grande. Alcuni di questi computer hanno più CPU che condividono una memoria comune. Altri invece sono costituiti da varie CPU, ma senza condivisione di memoria. Alcuni sono supercomputer, altri sono sistemi su singolo chip, mentre altri ancora sono COW.

Il Capitolo 9 contiene una lista commentata e organizzata per argomenti di letture consigliate, oltre a un elenco alfabetico di riferimenti bibliografici. È il capitolo più importante di tutto il libro. Lo si usi.

Problemi

1. Si spieghino con parole proprie i seguenti termini:
   a. traduttore (compilatore)
   b. interprete
   c. macchina virtuale.
2. Qual è la differenza tra interpretazione e traduzione?
3. Ha significato che un compilatore generi output per il livello di microarchitettura invece che per il livello ISA? Si analizzino i pro e i contro di tale ipotesi.
4. È possibile immaginare un computer multilivello in cui il livello dei dispositivi e i livelli logico digitali non siano i livelli più bassi? Si motivi la risposta.
5. Si consideri un computer in cui tutti i livelli siano diversi. Ciascun livello possiede istruzioni che sono m volte più potenti di quelle del livello sottostante; cioè un'istruzione del livello r può compiere il lavoro di m istruzioni del livello $r-1$. Se un programma del livello 1 richiede k secondi per essere eseguito, quanto tempo impiegheranno gli equivalenti programmi dei livelli 2, 3 e 4, assumendo che siano necessarie n istruzioni del livello r per interpretare una singola istruzione del livello $r+1$?
6. Alcune istruzioni del livello macchina del sistema operativo sono identiche alle istruzioni del linguaggio ISA. Queste istruzioni sono eseguite direttamente dal microprogramma invece che dal sistema operativo. Alla luce della risposta data al problema precedente, perché questo avviene?

7. Si consideri un computer con interpreti identici ai livelli 1, 2 e 3. Un interprete impiega *n* istruzioni per prelevare, esaminare ed eseguire un'istruzione. Se un'istruzione del livello 1 richiede *k* nanosecondi per essere eseguita, quanto tempo impiega un'istruzione ai livelli 2, 3 e 4?
8. In che senso l'hardware e il software sono equivalenti? E in che senso non lo sono?
9. La *difference engine* di Babbage aveva un unico programma, fisso, che non poteva essere modificato. È essenzialmente la stessa cosa di un moderno CD-ROM il cui contenuto non può essere cambiato? Si motivi la risposta.
10. Una delle conseguenze dell'idea di von Neumann di memorizzare i programmi in memoria è che anch'essi possono essere modificati, esattamente come i dati. È possibile immaginare un esempio in cui questa funzione potrebbe essere utile? (Suggerimento: si pensi a operazioni aritmetiche sugli array).
11. Il rapporto tra le prestazioni del 360 model 75 e del 360 model 30 era di 50 volte, mentre il ciclo di clock era solo 5 volte più veloce. Come si spiega questa differenza?
12. Nella Figura 1.5 e nella Figura 1.6 sono mostrati due elementari progetti di un sistema. Si descriva come potrebbero avvenire gli input e output in ciascuno di loro. Quale dei due può potenzialmente fornire migliori prestazioni generali del sistema?
13. Si supponga che ogni giorno ciascuno dei 300 milioni di cittadini americani consumi due confezioni di un certo articolo a cui è associata un'etichetta RFID. Quante etichette RFID devono essere prodotte annualmente per soddisfare la domanda? Qual è il costo totale delle etichette, se ognuna di loro costa un penny? Considerando l'ammontare del Prodotto Interno Lordo questa quantità di denaro potrebbe oppure no essere un ostacolo all'impiego di etichette RFID per ogni confezione messa in vendita?
14. Si citino tre elettrodomestici che potrebbero funzionare mediante una CPU integrata.
15. A un certo momento della storia un transistor su un microprocessore arrivò ad avere un diametro di 1 dm. Secondo la legge di Moore, che dimensioni avrebbe assunto nell'anno successivo?
16. La diatriba legale su chi ha inventato il computer fu risolta nell'aprile del 1973 dal giudice Earl Larson; egli si occupò della causa legale di violazione di brevetto intentata dalla Sperry Rand Corporation, società che aveva comprato i brevetti dell'elaboratore ENIAC. La posizione della Sperry Rand era che chiunque avesse prodotto un computer avrebbe dovuto pagarle i diritti, in quanto proprietaria dei brevetti chiave. Il caso arrivò in giudizio nel giugno 1971 e furono depositati più di 30.000 documenti. La trascrizione degli atti superò le 20.000 pagine. Utilizzando le vaste informazioni disponibili su Internet si studi il caso più approfonditamente e si scriva una relazione sui suoi aspetti tecnici. Che cosa sosteneva il brevetto di Eckert e Mauchley e perché il giudice ritenne che il loro sistema era basato sul precedente lavoro di Atanasoff?
17. Si scelgano le tre persone che si ritiene abbiano avuto la maggiore influenza nella creazione dell'hardware moderno; si scriva quindi una breve relazione che descriva il loro contributo e che motivi la scelta effettuata.
18. Si ripeta l'esercizio precedente nel caso del software.

Capitolo 2

Organizzazione dei sistemi di calcolo

Un calcolatore digitale è un sistema in cui processori, memorie e dispositivi periferici sono connessi tra loro. Questo capitolo fornisce un'introduzione a questi tre componenti e alle loro interconnessioni, servendo da base per l'analisi dettagliata dei singoli livelli che verrà data nei cinque capitoli successivi. Dato che i processori, le memorie e le periferiche sono concetti chiave che ritorneranno durante la trattazione di ciascun livello, iniziamo lo studio dell'architettura degli elaboratori esaminandoli uno alla volta.

2.1 Processori

La Figura 2.1 mostra l'organizzazione di un semplice calcolatore *bus-oriented*. La **CPU** (*Central Processing Unit*, "unità centrale di calcolo") è il "cervello" del computer e la sua funzione è quella di eseguire i programmi contenuti nella memoria principale prelevando le loro istruzioni, esaminandole ed eseguendole una dopo l'altra. I componenti sono connessi fra loro mediante un **bus**, cioè un insieme di cavi paralleli sui quali vengono trasmessi indirizzi, dati e segnali di controllo. I bus possono essere esterni alla CPU, per connetterla alla memoria e ai dispositivi di I/O, oppure interni, come vedremo tra poco.

La CPU è composta da parti distinte, tra cui l'unità di controllo e l'unità aritmetico-logica. La prima si occupa di prelevare le istruzioni dalla memoria principale e di determinarne il tipo, mentre la seconda esegue le operazioni, come l'addizione e l'AND, necessarie per portare a termine l'esecuzione delle istruzioni.

La CPU contiene anche una piccola memoria ad alta velocità, utilizzata per memorizzare i risultati temporanei e alcune informazioni di controllo. Questa memoria è costituita da un certo numero di registri, ciascuno dei quali ha una funzione e una dimensione predefinite. Ognuno di loro può contenere un numero, il cui valore può variare fino a un massimo che dipende dalla dimensione del registro, che di solito è uguale per tutti i registri. Dato che sono interni alla CPU possono essere letti e scritti a velocità elevate.

Il registro più importante è il **contatore d'istruzioni**, o **Program Counter** (**PC**), che punta alla successiva istruzione che dovrà essere prelevata per l'esecuzione (il termine "contatore d'istruzioni", pur essendo universalmente accettato, è in qualche modo ambiguo, dato che tale registro non effettua alcun *conteggio*). Un altro registro particolarmente importante è il **registro istruzione corrente**, o *Instruction Register* (**IR**), contenente l'istruzione che si

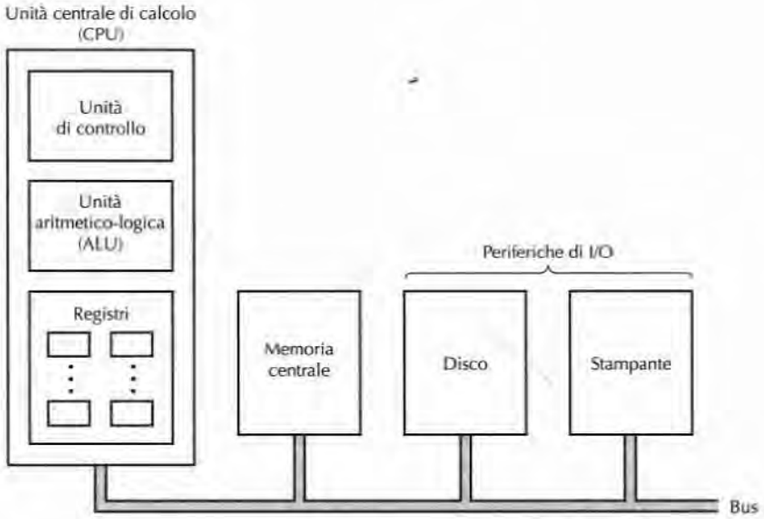


Figura 2.1 Organizzazione di un semplice calcolatore con una CPU e due periferiche di I/O.

trova in fase di esecuzione (la maggior parte dei calcolatori possiede molti altri registri, alcuni di uso generale, mentre altri sono preposti a un uso specifico).

2.1.1 Organizzazione della CPU

La Figura 2.2 mostra com'è organizzata internamente una parte, chiamata **percorso dati** (*data path*), di una tipica CPU di von Neumann; essa è composta dai registri (generalmente da 1 a 32), dalla **ALU** (*Arithmetic Logic Unit*, "unità aritmetico-logica") e da alcuni bus che connettono fra loro le diverse parti. I registri alimentano due registri di input della ALU (indicati nella figura con le lettere A e B) che mantengono i dati d'ingresso della ALU mentre questa è occupata nell'esecuzione di alcune computazioni. Il percorso dati riveste una grande importanza in tutte le macchine e nel corso del libro lo tratteremo in modo approfondito.

La ALU esegue alcune semplici operazioni sui suoi input, come addizioni e sottrazioni, e genera un risultato che viene memorizzato in un suo apposito registro di output. Questo valore può essere successivamente immagazzinato in uno dei registri della CPU che, volendo, può essere copiato in memoria in un secondo momento. Non tutti i tipi di progettazione prevedono la presenza dei registri A, B e di quello di output. Nell'esempio è illustrata l'addizione.

La maggior parte delle istruzioni può essere divisa in due categorie principali: le istruzioni registro-memoria e quelle registro-registro. Le istruzioni registro-memoria permettono di prelevare parole di memoria per portarle all'interno dei registri, dove sono utilizzabili, per esempio, come input della ALU per effettuare istruzioni successive. (Le "parole" sono le unità di dati che vengono spostate tra la memoria e i registri; una parola potrebbe essere un intero. Più avanti nel capitolo analizzeremo l'organizzazione della memoria.) Altre istruzioni registro-memoria permettono invece di copiare i valori dei registri nella memoria.

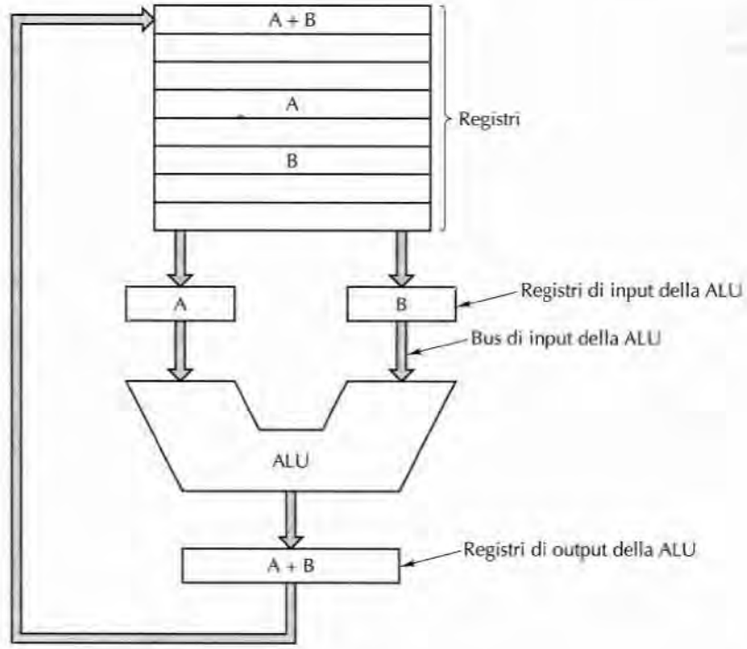


Figura 2.2 Percorso dati di una tipica macchina di von Neumann.

L'altra classe d'istruzioni è quella registro-registro. Una tipica istruzione di questo tipo preleva due operandi dai registri, li porta all'interno dei registri di input della ALU, esegue su di loro una qualche operazione (per esempio l'addizione o l'AND) e ne memorizza il risultato in uno dei registri. Il processo che consiste nel portare i due operandi attraverso la ALU e nel memorizzare il risultato è chiamato **ciclo del percorso dati** e rappresenta il cuore della maggior parte delle CPU. Con buona approssimazione si può dire che definisca che cosa sia in grado di fare una macchina; inoltre più veloce è il ciclo del percorso dati, maggiore risulta la velocità del calcolatore.

2.1.2 Esecuzione dell'istruzione

La CPU esegue ogni istruzione compiendo una serie di piccoli passi che, in linea generale, possono essere descritti nel seguente modo:

1. prelevare la successiva istruzione dalla memoria per portarla nell'IR
2. modificare il PC per farlo puntare all'istruzione seguente
3. determinare il tipo dell'istruzione appena prelevata
4. se l'istruzione usa una parola in memoria, determinare dove si trova

```
public class Interp {
   static int PC;                          // il program counter contiene l'indirizzo dell'istruzione successiva
   static int AC;                          // l'accumulatore, un registro per i calcoli aritmetici
   static int instr;                       // un registro che contiene l'istruzione corrente
   static int instr_type;                  // il tipo d'istruzione (opcode)
   static int data_loc;                    // l'indirizzo del dato, o -1 se non c'è
   static int data;                        // contiene l'operando corrente
   static boolean run_bit = true;          // un bit che può essere impostato a 0 per arrestare la macchina

   public static void interpret(int memory[ ], int starting_address {
      // Questa procedura interpreta programmi per una semplice macchina le cui istruzioni hanno
      // un solo operando in memoria. La macchina ha un registro AC (accumulatore), utilizzato per
      // i calcoli aritmetici. L'istruzione ADD, per esempio, somma un intero memorizzato in memoria al registro AC.
      // L'interprete continua finché il bit di esecuzione non viene impostato a 0 dall'istruzione HALT.
      // Lo stato di un processo eseguito su questa macchina è costituito dalla memoria, dal
      // contatore di programma, dal bit di esecuzione e dal registro AC. I parametri di input consistono
      // nell'immagine della memoria e nell'indirizzo di partenza.

      PC = starting_address;
      while (run_bit) {
         instr = memory[PC];               // preleva l'istruzione successiva e la memorizza in instr
         PC = PC + 1;                      // incrementa il program counter
         instr_type = get_instr_type(instr);  // determina il tipo d'istruzione
         data_loc = find_data(instr, instr_type);  // trova la posizione del dato (-1 se non c'è)
         if (data_loc >= 0)                // se "data_loc" vale -1, allora non c'è nessun operando
            data = memory[data_loc]        // preleva il dato
         execute(instr_type, data);        // esegue l'istruzione
      }

   }

   private static int get_instr_type(int addr) { ... }
   private static int find_data(int instr, int type) { ... }
   private static void execute(int type, int data) { ... }
}
```

Figura 2.3 Interprete Java per un semplice calcolatore.

5. se necessario, prelevare la parola per portarla in un registro della CPU
6. eseguire l'istruzione
7. tornare al punto 1 per iniziare l'esecuzione dell'istruzione successiva.

Spesso ci si riferisce a questa sequenza di passi con il termine di ciclo **esecutivo delle istruzioni**, o ciclo di **prelievo-decodifica-esecuzione**. Essa ha un'importanza centrale nel funzionamento di qualsiasi calcolatore.

La precedente descrizione del funzionamento della CPU assomiglia osservandola attentamente a un programma scritto in italiano. La Figura 2.3 mostra come sia possibile formalizzare questo programma in termini di un metodo (cioè una procedura) Java, che prende il nome di *interprete*. La macchina che viene interpretata possiede due registri visibili all'utente: il contatore di programma (PC), per tener traccia dell'indirizzo dell'istruzione successiva da prelevare, e l'accumulatore (AC), per contenere il risultato delle operazioni aritmetiche. Essa possiede anche alcuni registri interni per mantenere l'istruzione corrente durante la sua esecuzione (`instr`), il tipo d'istruzione corrente (`instr_type`), l'indirizzo dell'operando dell'istruzione (`data_loc`) e l'operando stesso (`data`). Si assume che le

istruzioni contengano un unico indirizzo di memoria e che nella locazione di memoria corrispondente sia memorizzato l'operando, che può essere, per esempio, un dato da sommare nell'accumulatore.

Il fatto che sia possibile scrivere un programma che simula il comportamento della CPU mostra che esso non deve per forza essere eseguito da una scatola piena di componenti elettronici, cioè dalla CPU fisica. Un programma può infatti essere portato a termine anche da un altro programma, che preleva, esamina ed esegue le proprie istruzioni; come abbiamo già detto nel corso del Capitolo 1, quest'ultimo programma viene chiamato **interprete**.

Questa equivalenza tra i processori hardware e gli interpreti ha alcune importanti implicazioni nell'organizzazione e nel progetto dei sistemi di calcolatori. Un gruppo di progettisti, dopo aver specificato il linguaggio macchina, *L*, per un nuovo calcolatore, può scegliere se realizzare un processore hardware capace di eseguire direttamente i programmi in *L* oppure scrivere un programma che interpreti tali programmi. Se si sceglie di scrivere un interprete occorre anche prevedere i componenti hardware che permettano di eseguirlo. È inoltre possibile avere delle costruzioni ibride, nelle quali alcune esecuzioni vengono svolte dall'hardware, mentre altre dall'interpretazione software.

Dato che un interprete scompone le istruzioni della propria macchina obiettivo in piccoli passi, il calcolatore sul quale viene eseguito può essere molto più semplice e meno costoso rispetto a quanto lo sarebbe un processore hardware appositamente progettato per la macchina obiettivo. Tale convenienza risulta ancor più significativa se la macchina obiettivo possiede molte istruzioni e se queste sono particolarmente complesse e caratterizzate da molte opzioni. Il vantaggio deriva essenzialmente dal fatto che si sostituisce parte dell'hardware con il software (l'interprete), molto meno costoso da riprodurre.

I primi calcolatori avevano un piccolo insieme di semplici istruzioni, ma la richiesta di calcolatori con prestazioni più elevate portò, fra le altre cose, all'introduzione d'istruzioni più potenti. In una fase iniziale si notò che spesso l'uso d'istruzioni più complesse aumentava la velocità di esecuzione dei programmi, anche se le singole istruzioni richiedevano più tempo per essere eseguite. Alcuni esempi di queste istruzioni più complesse comprendono quelle in virgola mobile e quelle per l'accesso diretto agli elementi di un array. In alcuni casi si osservò semplicemente che due istruzioni apparivano spesso consecutivamente e che quindi una singola istruzione avrebbe potuto svolgere il lavoro di entrambe.

Le istruzioni più complesse erano preferibili in quanto le singole esecuzioni delle operazioni elementari potevano essere sovrapposte oppure eseguite in parallelo utilizzando parti diverse dell'hardware. Dato che i calcolatori ad alte prestazioni giustificavano pienamente il costo di questo hardware aggiuntivo, essi arrivarono ad avere un numero molto maggiore d'istruzioni rispetto alle macchine di costo inferiore. Tuttavia, a causa della crescita del costo dello sviluppo del software e per via dei requisiti di compatibilità, sorse la necessità d'introdurre istruzioni complesse anche nelle macchine più economiche, per le quali il costo era più importante della velocità.

Alla fine degli anni '50 IBM (la società di computer più potente dell'epoca) comprese che mantenere una singola famiglia di macchine, tutte capaci di eseguire le stesse istruzioni, risultava molto vantaggioso non solo per IBM stessa, ma anche per i suoi clienti. IBM introdusse il termine **architettura** per descrivere questo livello di compatibilità. Una nuova famiglia di calcolatori avrebbe dovuto avere un'unica architettura, ma varie e distinte implementazioni che, pur distinguendosi nel prezzo e nelle prestazioni, sarebbero state in grado

di eseguire lo stesso programma. Com'è possibile tuttavia costruire una macchina economica capace di eseguire tutte le complicate istruzioni di macchine costose e ad alte prestazioni?

La risposta risiede nell'interpretazione. Questa tecnica, suggerita inizialmente da Wilkes (1951), ha permesso di progettare calcolatori semplici ed economici, in grado tuttavia di eseguire un gran numero d'istruzioni. Il risultato dell'interpretazione fu la nascita dell'architettura IBM System/360, una famiglia di calcolatori fra loro compatibili e che variavano di quasi due ordini di grandezza sia nel prezzo sia nelle prestazioni. Soltanto nei modelli più costosi si ricorse a un'implementazione diretta dell'hardware (cioè senza interpretazione).

I calcolatori semplici che utilizzano istruzioni interpretate presentano anche altri vantaggi, tra i quali:

1. la capacità di correggere agevolmente le istruzioni implementate in modo errato, o addirittura di compensare le mancanze progettuali dell'hardware di base
2. la possibilità di aggiungere nuove istruzioni a un costo minimo, anche dopo aver distribuito la macchina
3. un progetto strutturato che permetta di sviluppare, testare e documentare efficientemente le istruzioni complesse.

Negli anni '70, quando il mercato dei computer esplose in modo impressionante e la potenza di calcolo cominciò a crescere rapidamente, la richiesta di calcolatori a basso costo favorì la progettazione di macchine che usavano interpreti. La possibilità di adattare l'hardware e l'interprete per un determinato insieme d'istruzioni apparve come una modalità di progettazione dei processori altamente conveniente dal punto di vista dei costi. Dato che la tecnologia dei semiconduttori avanzava rapidamente, i vantaggi sui costi acquisirono un peso maggiore rispetto alla possibilità di avere prestazioni più elevate; le architetture basate sugli interpreti divennero così il modo convenzionale di progettare calcolatori. Quasi tutti i computer costruiti negli anni '70, dai minicomputer fino ai mainframe, si basavano infatti sull'interpretazione.

Nei tardi anni '70 divenne molto diffuso l'uso di processori semplici che eseguivano interpreti, fatta eccezione per i modelli più costosi e dalle prestazioni più elevate, come il Cray-1 e la serie Control Data Cyber. L'uso di un interprete eliminò i problemi di costo che avevano limitato l'uso d'istruzioni complesse e le architetture iniziarono a esplorare istruzioni ancor più complicate, soprattutto nel modo in cui venivano specificati gli operandi da utilizzare.

Questa tendenza raggiunse il suo apice con il calcolatore VAX della Digital Equipment Corporation, che aveva varie centinaia d'istruzioni e più di 200 modi diversi di specificare gli operandi da utilizzare in ciascuna istruzione. Sfortunatamente l'architettura VAX fu concepita fin dall'inizio per essere implementata attraverso un interprete e con poca attenzione alla realizzazione di un modello ad alte prestazioni. Questo approccio portò all'inclusione di un grandissimo numero d'istruzioni la cui importanza era marginale e che era difficile poter eseguire direttamente. Questo errore risultò fatale alla macchina VAX e, in ultima analisi, alla DEC stessa (nel 1998 Compaq acquistò DEC e nel 2001 Hewlett-Packard acquistò Compaq).

Se i primi microprocessori a 8-bit erano macchine con un insieme d'istruzioni molto semplice, alla fine degli anni '70 ogni microprocessore era passato invece a una progettazione basata sull'interpretazione. Durante questo periodo una delle maggiori sfide che si presentavano ai progettisti dei microprocessori era quella di gestire la crescente complessità resa possibile dai circuiti integrati. Uno dei maggiori vantaggi dell'approccio basato sull'interpretazione era la possibilità di progettare un processore semplice, la cui complessità era in gran parte confinata nell'interprete contenuto in memoria; in questo modo era possibile portare la complessità progettuale dall'hardware al software.

Il successo del Motorola 68000, che disponeva di un ampio insieme d'istruzioni interpretate, e il contemporaneo fallimento dello Zilog Z8000 (che aveva anch'esso un grande insieme d'istruzioni, ma senza interprete) mostrò quali vantaggi offriva un interprete nel mettere velocemente sul mercato un nuovo microprocessore. Questo successo fu ancora più sorprendente se si considera il vantaggio che all'epoca aveva la società Zilog (il processore Z80, predecessore dello Z8000, era stato infatti molto più diffuso del 6800, predecessore del 68000). Ovviamente vi furono anche altre concause, non ultima la lunga esperienza di Motorola come produttore di processori e quella di Exxon (la società che controllava la Zilog) come società petrolifera, e non certo come produttore di processori.

Un altro fattore che all'epoca favorì l'interpretazione fu l'esistenza di veloci memorie di sola lettura, chiamate **memorie di controllo** (*control store*), usate per memorizzare l'interprete. Supponiamo che una normale istruzione interpretata richiedesse 10 istruzioni dell'interprete, chiamate **microistruzioni**, ciascuna da 100 ns, oltre a due riferimenti alla memoria centrale da 500 ns l'uno. L'esecuzione totale richiedeva quindi 2000 ns, un valore solo due volte peggiore del miglior risultato raggiungibile tramite l'esecuzione diretta; senza l'utilizzo della memoria di controllo, le istruzioni avrebbero invece impiegato 6000 ns. È evidente che un fattore di penalizzazione di sei volte è molto più difficile da accettare rispetto a un fattore due.

2.1.3 RISC contro CISC

Durante i tardi anni '70 gli interpreti permisero di sperimentare istruzioni molto complesse; il tentativo inseguito dai progettisti era quello di ridurre il "gap semantico" che divideva ciò che erano in grado di fare le macchine da quello che invece richiedevano i linguaggi di programmazione ad alto livello. Quasi mai si pensava di progettare macchine più semplici, esattamente come oggi non sono molti i ricercatori che si impegnano nel progettare sistemi operativi, reti, elaboratori di testi, ecc. meno potenti (e forse ciò è un peccato).

Ciononostante un gruppo guidato da John Cocke di IBM andò contro questa tendenza e provò a implementare alcune delle idee di Seymour Cray all'interno di un minicomputer ad alte prestazioni. Il risultato di questo lavoro fu un minicomputer sperimentale, chiamato **801**[1]. Anche se IBM non mise mai in commercio questa macchina e i risultati furono pubblicati soltanto alcuni anni dopo (Radin, 1982), trapelarono alcune voci riguardanti il progetto e altri ricercatori iniziarono a studiare architetture simili. Nel 1980, a Berkley, un gruppo guidato da David Patterson e Carlo Séquin cominciò la progettazione di chip VLSI per CPU che non utilizzavano l'interpretazione (Patterson, 1985; Patterson e Séquin, 1982). Per indicare questo progetto essi coniarono l'acronimo **RISC** e chiamarono la loro CPU RISC I, che fu seguita a breve dal modello RISC II. Poco dopo, nel 1981,

[1] Pare che questo nome derivi dal numero dell'edificio del centro IBM in cui si svolse il progetto (*N.d.Rev.*).

a Stanford, sull'altra sponda della baia di San Francisco, John Hennessy progettò e costruì un chip, chiamato **MIPS**, leggermente differente sotto alcuni aspetti (Hennessy, 1984). Entrambi i chip evolsero in due importanti prodotti commerciali, rispettivamente SPARC e MIPS.

Questi nuovi processori erano significativamente diversi rispetto a quelli in commercio all'epoca. Dato che non dovevano essere retrocompatibili con alcun prodotto esistente, i loro progettisti furono liberi di scegliere nuovi insiemi d'istruzioni che massimizzassero le prestazioni generali del sistema. Mentre l'enfasi iniziale fu posta su istruzioni semplici la cui esecuzione potesse essere completata velocemente, presto si capì che la chiave per ottenere buone prestazioni consisteva nel progettare istruzioni che potessero essere **emesse** (iniziate) velocemente. Il tempo impiegato effettivamente per l'esecuzione di un'istruzione aveva un'importanza minore rispetto a quante se ne potevano iniziare in un secondo.

Quando vennero progettati questi semplici processori, la caratteristica che colpì maggiormente l'attenzione di tutti fu che il numero d'istruzioni disponibili era relativamente basso, generalmente attorno a 50. Questo valore era nettamente inferiore alle 200-300 dei computer convenzionali come il VAX e i grandi mainframe IBM. L'acronimo RISC significa infatti **computer con un insieme ridotto d'istruzioni** (*Reduced Instruction Set Computer*), scelto in contrapposizione a CISC, che significa **computer con un insieme d'istruzioni complesso** (*Complex Instruction Set Computer*): un sottile riferimento al VAX, che aveva dominato i dipartimenti universitari di quegli anni. L'uso del nome si è radicato ed è utilizzato ancora oggi, sebbene attualmente siano in pochi a ritenere che la dimensione dell'insieme d'istruzioni sia un problema importante.

In sintesi si può affermare che si scatenò una guerra di religione che vedeva i sostenitori di RISC attaccare l'ordine prestabilito (VAX, Intel e i grandi mainframe IBM). Essi sostenevano che il miglior modo per progettare un calcolatore fosse quello di avere poche istruzioni semplici che potevano essere eseguite in un ciclo del percorso dati mostrato nella Figura 2.2, per esempio leggendo due registri, combinandoli in qualche modo (come sommandoli o calcolandone il prodotto logico) e memorizzando infine il risultato all'interno di un registro. La loro tesi era che una macchina RISC prevaleva nei confronti di una CISC perché, anche se impiegava quattro o cinque istruzioni per fare ciò che una macchina CISC realizzava con una sola, le sue istruzioni erano 10 volte più veloci (in quanto non interpretate). Vale la pena sottolineare che a quel tempo la velocità delle memorie centrali aveva raggiunto quella delle memorie di controllo di sola lettura, rendendo ancora più marcata la penalizzazione dovuta all'interpretazione; ciò contribuì a favorire fortemente le macchine RISC.

Si potrebbe pensare che, grazie ai vantaggi offerti dalla tecnologia RISC, queste macchine (come la Sun UltraSPARC) abbiano fatto scomparire dal mercato le macchine CISC (come gli Intel Pentium), e invece ciò non è avvenuto: perché?

Primo fra tutti va considerato il problema della retrocompatibilità e con esso i miliardi di dollari investiti dalle società per produrre il software per le macchine Intel. Secondo, Intel è riuscita sorprendentemente a impiegare le stesse idee anche all'interno dell'architettura CISC. A partire dal processore 486, le CPU di Intel contengono un nucleo di tipo RISC che esegue le istruzioni più semplici (e spesso più comuni) in un unico ciclo del percorso dati, mentre interpreta le istruzioni più complesse secondo la classica modalità CISC. Il risultato finale è che le istruzioni più comuni sono veloci, mentre quelle meno comuni sono più lente.

Anche se questo approccio non è altrettanto veloce quanto una pura architettura RISC, esso fornisce prestazioni generali competitive, permettendo allo stesso tempo di eseguire senza alcuna modifica il software progettato precedentemente.

2.1.4 Principi di progettazione dei calcolatori moderni

Ora che sono passati più di due decenni dall'introduzione delle prime macchine RISC, alcuni principi di progettazione sono stati accettati come un buon modo di progettare un calcolatore, tenendo in considerazione lo stato attuale della tecnologia dell'hardware. Se dovesse verificarsi un importante cambiamento tecnologico (per esempio un nuovo processo di produzione tale da rendere il ciclo della memoria 10 volte più veloce di quello della CPU), molte delle scommesse finora fatte risulterebbero perdenti. I progettisti di calcolatori devono quindi tenere sempre d'occhio i cambiamenti tecnologici, in quanto possono modificare l'equilibrio fra i vari componenti.

Detto questo, esiste un insieme di principi di progettazione, talvolta chiamati **principi di progettazione RISC**, che i progettisti delle CPU cercano di seguire il più possibile. Anche se vincoli esterni, come i requisiti di retrocompatibilità con alcune architetture esistenti, richiedono di tanto in tanto alcuni compromessi, questi principi sono comunque degli obiettivi che i progettisti si sforzano di raggiungere. Di seguito ne analizzeremo i principali.

Tutte le istruzioni sono eseguite direttamente dall'hardware

Tutte le comuni istruzioni sono eseguite direttamente dall'hardware, e non sono interpretate mediante microistruzioni. L'eliminazione di un livello di interpretazione garantisce velocità più alte per la maggior parte delle istruzioni. Per i calcolatori che implementano insiemi d'istruzioni CISC, quelle più complesse devono essere divise in varie parti che possono essere eseguite come una sequenza di microistruzioni. Questo passo aggiuntivo rallenta la macchina, ma può tuttavia essere accettabile per le istruzioni utilizzate con minor frequenza.

Massimizzare la frequenza di emissione delle istruzioni

I calcolatori moderni ricorrono a molti trucchi per massimizzare le loro prestazioni, il principale dei quali consiste nel cercare di iniziare a eseguire più istruzioni al secondo. Dopo tutto, se si riescono a emettere 500 milioni d'istruzioni al secondo, si ottiene un processore da 500 MIPS, indipendentemente da quanto tempo impieghino quelle istruzioni per essere completate[2]. Questo principio suggerisce che il parallelismo può giocare un ruolo importante nelle prestazioni; infatti è possibile emettere un gran numero di lente istruzioni in un breve intervallo di tempo solo se si riescono a eseguire più istruzioni allo stesso tempo.

Anche se le istruzioni vengono sempre incontrate nell'ordine in cui appaiono nel programma, non sempre sono emesse rispettando questa sequenza (dato che alcune risorse necessarie potrebbero essere occupate) e non è neanche necessario che terminino in tale ordine. Ovviamente, se l'istruzione 1 imposta il valore di un registro e l'istruzione 2 usa lo

---

[2] Tra l'altro, **MIPS** – acronimo di *Millions of Instructions Per Second* – sta per "milioni d'istruzioni al secondo", e il processore MIPS fu così chiamato in questo modo come gioco di parole in relazione all'acronimo (*N.d.Rev.*).

stesso registro, occorre prestare grande attenzione per evitare che l'istruzione 2 lo legga prima che contenga il giusto valore. Per far ciò in modo corretto è necessario tener conto di molti aspetti; ciononostante questo approccio garantisce un guadagno nelle prestazioni, dato che è possibile eseguire più istruzioni allo stesso tempo.

Le istruzioni devono essere facili da decodificare

Un limite critico sulla frequenza di emissione delle istruzioni è dato dal processo di decodifica, che deve essere effettuato per ogni singola istruzione allo scopo di determinare le risorse necessarie. Tutto ciò che può aiutare questo processo si rivela utile: per esempio rendere le istruzioni regolari, di lunghezza fissa e con pochi campi. Meno formati d'istruzioni ci sono, meglio è.

Solo le istruzioni Load e Store fanno riferimento alla memoria

Uno dei modi più semplici per spezzare le operazioni in passi separati è quello di richiedere che, per la maggior parte delle istruzioni, gli operandi vengano prelevati dai registri e siano memorizzati al loro interno. L'operazione di spostamento degli operandi dalla memoria ai registri può essere invece compiuta separatamente mediante apposite istruzioni. Dato che l'accesso alla memoria può richiedere un tempo considerevole, il cui ritardo non è prevedibile, queste operazioni possono essere efficientemente sovrapposte all'esecuzione di altre istruzioni. Tale osservazione porta alla conclusione che soltanto le istruzioni `LOAD` e `STORE` dovrebbero far riferimento alla memoria, mentre tutte le altre dovrebbero operare esclusivamente sui registri.

Molti registri disponibili

Dato che l'accesso alla memoria è relativamente lento occorre prevedere molti registri (almeno 32) di modo che, una volta prelevata la parola, possa essere mantenuta nel registro fintanto sia necessario. È particolarmente inefficiente trovarsi senza registri liberi, in quanto ciò obbliga a scaricare in memoria tutti i valori dei registri per poi ricaricarli. Il miglior modo per evitare il più possibile questa operazione consiste nel disporre di un numero sufficiente di registri.

2.1.5 Parallelismo a livello d'istruzione

I progettisti di calcolatori si sforzano costantemente di migliorare le prestazioni delle loro macchine. Aumentare la velocità di clock per rendere i processori più veloci è una possibilità, ma per qualsiasi nuova architettura esiste un limite, dipendente dal momento storico, su che cosa sia possibile ottenere mediante la semplice forza bruta. Per questo motivo molti progettisti di computer vedono nel parallelismo (cioè nel compiere più azioni allo stesso tempo) un modo per ottenere prestazioni più elevate con una data velocità di clock.

Il parallelismo può essere presente in due forme principali: a livello d'istruzione e a livello di processore. Nella prima forma il parallelismo è sfruttato all'interno delle singole istruzioni per far sì che la macchina possa elaborarne un maggior numero al secondo, mentre nella seconda sono presenti più CPU che lavorano congiuntamente su uno stesso problema. Entrambi gli approcci hanno i propri pregi; in questo paragrafo analizzeremo il primo tipo di parallelismo, mentre nel prossimo studieremo quello a livello di processore.

Pipelining

Da anni ormai è assodato che uno dei maggiori colli di bottiglia nella velocità di esecuzione delle istruzioni è rappresentato dal prelievo delle istruzioni dalla memoria. Fin dal modello IBM Stretch (1959), per alleviare questo problema, i calcolatori sono stati dotati della capacità di poter prelevare in anticipo le istruzioni dalla memoria, in modo da averle già a disposizione nel momento in cui dovessero rendersi necessarie. Le istruzioni venivano memorizzate in un insieme di registri chiamati **buffer di prefetch**, dai quali potevano essere prese nel momento in cui venivano richieste, senza dover attendere che si completasse una lettura della memoria.

In pratica la tecnica di *prefetching* divide l'esecuzione dell'istruzione in due parti: il prelievo dell'istruzione e la sua esecuzione effettiva. Il concetto di **pipeline** spinge questa strategia molto più avanti; invece di dividere l'esecuzione di un'istruzione solamente in due fasi, la si divide in un numero maggiore di parti (spesso una dozzina o più) che possono essere eseguite in parallelo; ciascuna di queste parti è gestita da componenti hardware dedicati.

La Figura 2.4(b) illustra il modello di pipeline. Durante il primo ciclo di clock lo stadio S1 sta lavorando sull'istruzione 1, prelevandola dalla memoria. Durante il ciclo di clock 2 lo stadio S2 decodifica l'istruzione 1, mentre lo stadio S1 preleva l'istruzione 2. Durante il ciclo 3 lo stadio S3 preleva gli operandi per l'istruzione 1, lo stadio S2 decodifica l'istruzione 2 e lo stadio S1 preleva la terza istruzione. Durante il quarto ciclo lo stadio S4 esegue l'istruzione 1, S3 preleva gli operandi per l'istruzione 2, S2 decodifica l'istruzione 3 e S1 preleva l'istruzione 4. Infine, durante l'ultimo ciclo S5 scrive il risultato dell'istruzione 1, mentre gli altri componenti lavorano sulle istruzioni successive.

Per rendere più chiaro il concetto di *pipelining* consideriamo un'analogia. Immaginiamo una fabbrica di dolciumi in cui i reparti di cottura e di confezionamento sono separati.

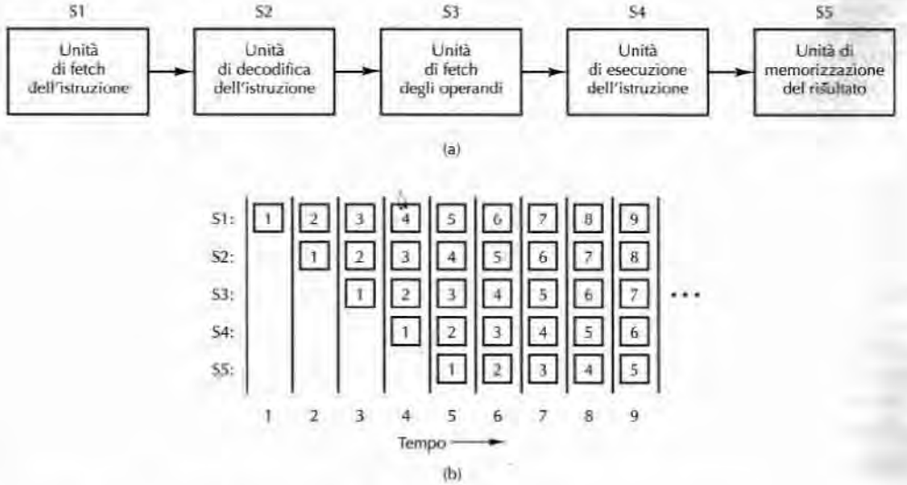


Figura 2.4 (a) Pipeline a cinque stadi. (b) Lo stato degli stadi in funzione del tempo. Sono mostrati nove cicli di clock.

Supponiamo che nel reparto spedizioni ci siano cinque operai (le unità di elaborazione) allineati lungo un nastro trasportatore. Ogni 10 secondi (il ciclo di clock), il primo operaio mette sul nastro una scatola vuota. La scatola viene passata al secondo operaio che inserisce al suo interno una torta. Poco dopo la scatola arriva alla postazione successiva dove viene chiusa e sigillata. Poi la scatola giunge al quarto operaio che vi attacca un'etichetta. Infine l'ultimo operaio rimuove la scatola dal nastro e la ripone in un grande contenitore per la spedizione a un supermercato. In sostanza questo è il modo in cui funziona la pipeline di un calcolatore: ogni istruzione (torta) attraversa vari passi di elaborazione prima di uscire, una volta completata, all'estremità opposta.

Tornando alla Figura 2.4 supponiamo che il ciclo di clock della macchina sia di 2 ns e che quindi un'istruzione impieghi 10 ns per percorrere i cinque stadi della pipeline. A prima vista, si potrebbe dire che la macchina abbia una velocità di 100 MIPS, ma in realtà le sue prestazioni sono molto migliori. Dato che in ogni ciclo di clock (2 ns) viene completata una nuova istruzione, la velocità reale di elaborazione è di 500 MIPS e non di soli 100.

L'uso della pipeline permette di bilanciare la **latenza** (il tempo che un'istruzione impiega per essere elaborata) e la **larghezza di banda del processore** (i MIPS della CPU). Con un ciclo di clock di T ns e una pipeline a n stadi, la latenza è di nT ns, poiché ogni istruzione attraversa n stadi, ognuno dei quali richiede T ns.

Dato che a ogni ciclo di clock viene completata un'istruzione e visto che vi sono $10^9/T$ cicli di clock al secondo, il numero d'istruzioni eseguite al secondo è $10^9/T$. Se per esempio $T = 2$ ns, vengono eseguite 500 milioni d'istruzioni al secondo. Dividendo la velocità di esecuzione delle istruzioni per un milione si ottiene il numero di MIPS, cioè $(10^9/T)/10^{-6} = 1000/T$ MIPS. In teoria potremmo misurare la velocità di esecuzione delle istruzioni in BIPS[3], al posto di MIPS, ma dato che nessuno lo fa non lo faremo neanche noi.

Architetture superscalari

Se è bene avere una pipeline, averne due è sicuramente meglio. La Figura 2.5 mostra un ipotetico progetto di una CPU con due pipeline, entrambe basate sullo schema della Figura 2.4. In questa situazione una singola unità di *fetch* preleva due istruzioni alla volta e le inserisce nelle pipeline, ognuna delle quali è dotata di una ALU. Affinché le due istruzioni possano essere eseguite in parallelo, non devono però esserci conflitti nell'uso delle risorse (cioè i registri) e nessuna delle due istruzioni deve dipendere dal risultato dell'altra. Come nel caso della singola pipeline, o è il compilatore a occuparsi di gestire correttamente questa situazione (l'hardware non effettua quindi alcun controllo e se le istruzioni sono incompatibili restituisce un risultato errato) oppure i conflitti sono rilevati ed eliminati durante l'esecuzione per mezzo di componenti hardware ad hoc.

Anche se le pipeline, singole o doppie, sono principalmente usate sulle macchine RISC, a partire dal 486 (il 386 e i suoi predecessori ne erano privi), Intel ha cominciato a introdurre le pipeline anche nelle proprie CPU. Il 486 aveva una sola pipeline, mentre il primo Pentium ne aveva due, entrambe a cinque stadi e simili a quelle mostrate nella Figura 2.5, sebbene la divisione esatta del lavoro tra gli stadi 2 e 3 (chiamati *decode-1* e *decode-2*) fosse leggermente diversa rispetto al nostro esempio. La pipeline principale, chiamata **pipeline u**,

[3] Qui la B sta per *Billions*, cioè miliardi (*N.d.T.*).

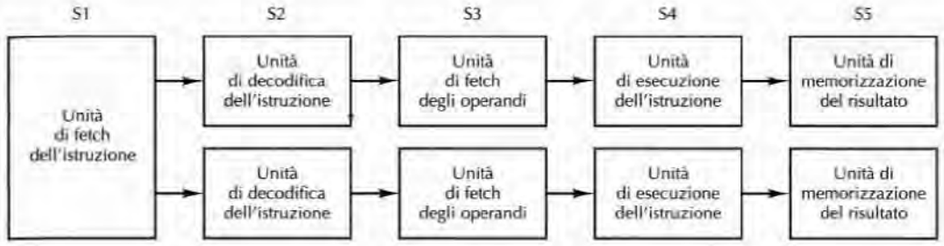


Figura 2.5 Doppia pipeline a cinque stadi con l'unità di *fetch* dell'istruzione in comune.

poteva eseguire una qualsiasi istruzione Pentium. La seconda pipeline, chiamata **pipeline v**, poteva invece eseguire solamente semplici istruzioni su interi (oltre a una semplice istruzione in virgola mobile, FXCH).

Alcune regole prestabilite determinavano se due istruzioni potevano essere eseguite in parallelo. Se tali istruzioni non erano sufficientemente semplici oppure erano incompatibili, soltanto la prima delle due veniva eseguita (nella pipeline u), mentre la seconda veniva trattenuta per essere successivamente accoppiata all'istruzione seguente. Le istruzioni erano sempre eseguite in ordine; per questo motivo compilatori specifici per il Pentium, che erano in grado di produrre coppie d'istruzioni compatibili, potevano generare programmi la cui esecuzione era più veloce rispetto a quella dei vecchi compilatori. Nel caso di programmi che lavoravano su interi, varie misurazioni mostrarono che su un Pentium il codice appositamente ottimizzato per la sua architettura veniva eseguito esattamente due volte più velocemente che su un 486 funzionante alla stessa velocità di clock (Pountain, 1993). Questo guadagno era da attribuire interamente alla seconda pipeline.

Anche se si potrebbe immaginare un'architettura a quattro pipeline, ciò obbligherebbe a duplicare troppi componenti hardware (gli informatici non credono nella magia del numero tre); per questo motivo nelle CPU di gamma alta si usa un approccio di tipo diverso. L'idea base consiste nell'avere una singola pipeline, ma di associarle più unità funzionali, com'è mostrato nella Figura 2.6. Il Pentium II, per esempio, ha una struttura simile a quella della figura e sarà esaminata nel corso del Capitolo 4. Nel 1987 fu coniato il termine **architettura superscalare** (Agerwala e Cocke, 1987) per indicare questo approccio; le sue radici risalgono però al calcolatore CDC 6600, ideato più di 40 anni prima. Il 6600 prelevava un'istruzione ogni 100 ns e la passava a una delle 10 unità funzionali che lavoravano in parallelo mentre la CPU era occupata a lanciare l'istruzione successiva.

Nel corso del tempo la definizione di "superscalare" si è in qualche modo evoluta; ora è utilizzata per descrivere processori che lanciano più istruzioni (spesso quattro o sei) durante un ciclo di clock. Ovviamente una CPU superscalare, per poter gestire tutte queste istruzioni, deve avere più unità funzionali. Dato che i processori superscalari hanno generalmente una sola pipeline, essi assomigliano allo schema della Figura 2.6.

Se si utilizza questa definizione, allora da un punto di vista tecnico il 6600 non era superscalare, in quanto lanciava una sola istruzione ogni ciclo. Tuttavia il risultato finale era praticamente lo stesso: le istruzioni venivano lanciate a una frequenza molto più elevata rispetto a quella a cui potevano essere eseguite. Vi è una differenza molto piccola tra una CPU con ciclo di clock di 100 ns che lancia un'istruzione per ciclo a un gruppo di unità funziona-

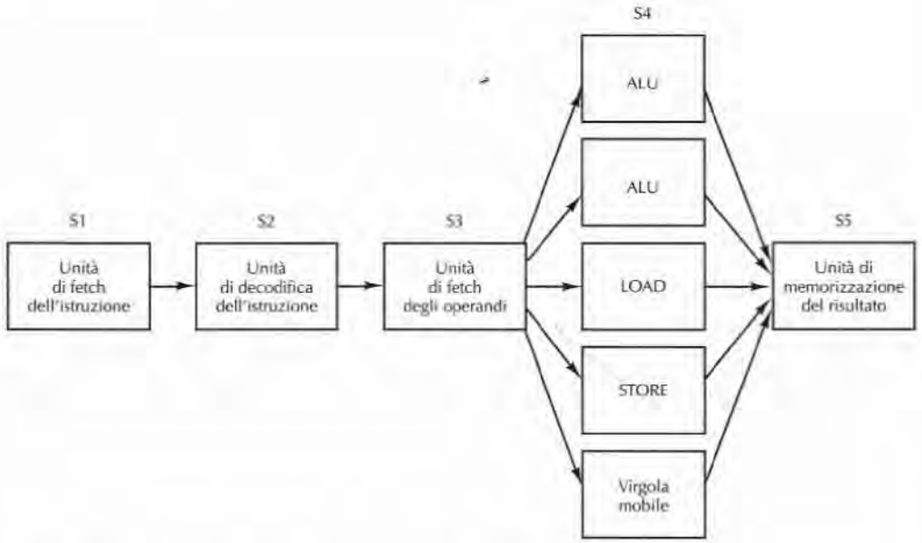


Figura 2.6 Processore superscalare con cinque unità funzionali.

li e una CPU con ciclo di clock di 400 ns che lancia allo stesso gruppo di unità funzionali quattro istruzioni per ciclo. In entrambi i casi l'idea chiave è che il tasso di lancio è molto più elevato del tasso di esecuzione, con il carico di lavoro che viene distribuito su un gruppo di unità funzionali.

Nell'idea di un processore superscalare è implicito il fatto che lo stadio S3 può lanciare istruzioni a una velocità significativamente superiore rispetto a quella a cui lo stadio S4 riesce a eseguirle. Se lo stadio S3 lanciasse un'istruzione ogni 10 ns e tutte le unità funzionali potessero compiere il loro lavoro in 10 ns, allora non più di un'istruzione alla volta sarebbe in esecuzione, facendo così crollare l'idea base. In realtà la maggior parte delle unità funzionali dello stadio 4 richiede un tempo di esecuzione significativamente maggiore di un singolo ciclo di clock, e ciò è in particolare vero per quelle che accedono alla memoria o che compiono calcoli aritmetici in virgola mobile. Come si può vedere nella figura è possibile avere più unità ALU nello stadio S4.

2.1.6 Parallelismo a livello di processore

La richiesta di calcolatori sempre più veloci sembra inarrestabile. Gli astronomi vogliono simulare che cosa successe durante il primo microsecondo successivo al big bang, gli economisti vogliono modellare l'economia su scala mondiale e gli adolescenti vogliono giocare su Internet con i loro amici virtuali con giochi 3D multimediali e interattivi. Dato che le CPU continuano a diventare più veloci, prima o poi ci si scontrerà con i problemi legati alla velocità della luce, il cui ritardo di propagazione è di 20 cm/ns sia nei cavi di rame sia nelle fibre ottiche, indipendentemente dall'abilità degli ingegneri di Intel. Inoltre, chip più veloci producono anche più calore, e la sua dissipazione costituisce un problema.

Il parallelismo a livello d'istruzione aiuta in parte, ma difficilmente l'uso della pipeline e le operazioni superscalari possono aumentare le prestazioni di un fattore cinque o dieci. Per ottenere guadagni di 50, 100, e più, l'unica soluzione è quella di progettare calcolatori con più CPU; per questo motivo analizzeremo ora come sono organizzati alcuni di questi sistemi.

Array computer

Un numero rilevante di problemi dell'ingegneria e della fisica riguarda strutture che presentano una forte regolarità e spesso gli stessi calcoli devono essere eseguiti allo stesso tempo su vari e distinti insiemi di dati. La regolarità della struttura di questi programmi li rende particolarmente adatti per essere accelerati mediante l'esecuzione parallela. Per eseguire velocemente programmi scientifici di grandi dimensioni sono stati impiegati sostanzialmente due diversi metodi. Sebbene questi due schemi siano molto simili sotto vari aspetti, stranamente uno è visto come un'estensione di un singolo processore, mentre l'altro come un calcolatore parallelo.

Un **array computer** (ovvero un array di processori) consiste di un gran numero di processori identici che eseguono la stessa sequenza d'istruzioni su insiemi diversi di dati. Il primo di questi prodotti fu il calcolatore ILLIAC IV della University of Illinois, illustrato nella Figura 2.7 (Bouknight et al., 1972). Il progetto originale prevedeva di costruire un macchina costituita da quattro quadranti, ciascuno formato da una griglia quadrata di 8 coppie di processori e memorie per lato. Una singola unità di controllo per ogni quadrante trasmetteva in broadcast le istruzioni, che venivano eseguite a passi sincronizzati da tutti i processori, ciascuno dei quali utilizzava i dati letti dalla propria memoria (caricata durante la fase d'inizializzazione). Talvolta si utilizza il termine processore **SIMD** (*Single Instruction-stream Multiple Datastream*, "istruzioni singole, dati multipli") per indicare questo tipo di architettura, nettamente diversa da una comune macchina di Von Neumann. Anche se venne costruito un solo quadrante per limitarne i costi, le prestazioni di ILLIAC IV raggiunsero i 50 megaflop (milioni di operazioni in virgola mobile al secondo). Si dice che, se la macchina fosse stata costruita interamente e se le sue prestazioni avessero raggiunto l'obiettivo origi-

Figura 2.7 Array di processori del tipo ILLIAC IV.

nario (1 gigaflop), essa avrebbe avuto una potenza computazionale pari a due volte quella del mondo intero.

Agli occhi di un programmatore, un **processore vettoriale** risulta molto simile a un array di processori. Come questi, esso esegue in modo molto efficiente una stessa sequenza di operazioni su coppie di dati, anche se, a differenza dagli array computer, tutte le operazioni di addizione sono eseguite da un unico sommatore, altamente strutturato a pipeline. L'azienda fondata da Seymour Cray, la Cray Research (che oggi fa parte di SGI), ha prodotto un gran numero di processori vettoriali, a partire dal modello Cray-1 nel 1974, fino ai modelli attuali.

Sia gli array di processori sia i processori vettoriali lavorano su array di dati. Entrambi eseguono singole istruzioni che, per esempio, sommano a coppie gli elementi di due vettori ma, mentre un array di processori lo fa usando tanti sommatori quanti sono gli elementi dei vettori, in un processore vettoriale si utilizza invece un **registro vettoriale**, che consiste di un insieme di registri convenzionali caricabili dalla memoria in una singola istruzione (che, in realtà, li carica in modo sequenziale). Un'istruzione di somma tra vettori viene quindi eseguita su coppie di elementi prelevati da due registri vettoriali per alimentare un sommatore strutturato a pipeline. Il risultato è un altro vettore che può essere memorizzato in un registro vettoriale oppure utilizzato direttamente come operando per una nuova operazione vettoriale.

Anche se oggi non vi sono array computer in produzione, l'idea di questo tipo di macchine non è morta. Le istruzioni MMX ed SSE del Pentium 4 utilizzano questo modello di esecuzione per velocizzare i programmi multimediali; sotto questo aspetto il calcolatore ILLIAC IV va considerato come uno degli antenati del Pentium 4.

Multiprocessori

In un array di processori le unità di elaborazione non sono delle CPU indipendenti, dato che c'è un'unica unità di controllo condivisa fra tutte. Il primo sistema parallelo che analizziamo e che è composto da più CPU complete è il **multiprocessore**, cioè un sistema composto da più CPU con una memoria in comune (così come un gruppo di persone condivide la stessa lavagna). Dato che ogni CPU può leggere e scrivere una qualsiasi parte della memoria, esse devono coordinarsi (via software) per evitare di ostacolarsi a vicenda. Quando due o più CPU hanno la possibilità di interagire in modo così profondo si dice che sono *tighly coupled* (cioè legate strettamente).

Sono possibili vari schemi d'implementazione, il più semplice dei quali consiste nell'avere un singolo bus con più CPU, tutte connesse a un'unica memoria. La Figura 2.8(a) mostra un diagramma di un simile multiprocessore.

Non serve una gran fantasia per capire che si verificano dei conflitti se un gran numero di processori veloci tenta costantemente di accedere alla memoria attraverso lo stesso bus. Per ridurre queste contese e migliorare le prestazioni i progettisti di multiprocessori hanno ideato vari schemi. La Figura 2.8 mostra un'architettura in cui ogni processore possiede una propria memoria locale, non accessibile agli altri. Questa memoria è utilizzabile per contenere il codice del programma e quei dati che non devono essere condivisi. L'accesso a questa memoria privata non utilizza il bus principale, riducendo quindi in modo considerevole il traffico sul bus. Oltre a questo schema ne esistono anche degli altri (per esempio, quello detto *caching*).

Rispetto ad altri tipi di calcolatori paralleli i multiprocessori hanno il vantaggio che è facile lavorare con un modello di programmazione basato su una memoria condivisa. Si im-

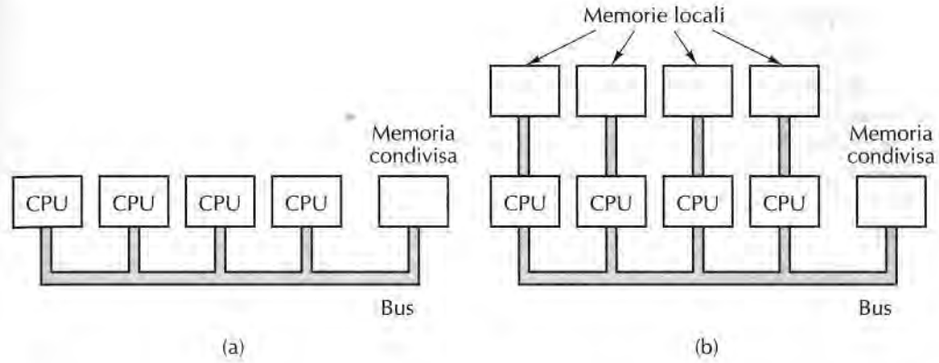


Figura 2.8 (a) Multiprocessore a bus singolo. (b) Multicomputer con memorie locali.

magini per esempio un programma che ricerca le cellule tumorali in un'immagine di un certo tessuto presa al microscopio. La fotografia digitalizzata potrebbe essere mantenuta in memoria comune e a ciascun processore potrebbe essere assegnata una particolare regione dell'immagine al cui interno effettuare la ricerca. Dato che ciascun processore ha accesso all'intera memoria, non vi è alcun problema se lo studio di una cellula, che inizia nella regione a esso assegnata, scavalca il limite della regione successiva.

Multicomputer

Se da un lato è relativamente semplice costruire multiprocessori composti da un modesto numero di processori (non più di 256), è invece decisamente più complicato realizzarne di più grandi. La difficoltà risiede nel connettere tutti i processori alla memoria. Per aggirare questi problemi molti progettisti hanno semplicemente abbandonato l'idea di avere una memoria condivisa e hanno costruito sistemi composti da un gran numero di calcolatori interconnessi, ciascuno dotato di una memoria privata. Questi sistemi sono detti **multicomputer**. In questi sistemi le CPU sono dette con legame lasco (*loosely coupled*), in contrapposizione con quelle che compongono i multiprocessori.

Le CPU dei multicomputer comunicano fra loro inviandosi messaggi, simili alle e-mail, ma molto più veloci. Nel caso di grandi sistemi, dato che non è efficiente connettere mutualmente tutti i calcolatori, si utilizzano topologie diverse come griglie 2D e 3D, alberi e anelli. Ne consegue che i messaggi tra due calcolatori per spostarsi dalla sorgente alla destinazione spesso devono passare attraverso uno o più macchine intermedie oppure attraverso dei commutatori. Ciononostante è possibile ottenere che lo scambio di messaggi richieda un tempo nell'ordine di pochi microsecondi. Sono stati costruiti e resi operativi multicomputer dotati di quasi 10.000 CPU.

Visto che è facile programmare i multiprocessori, mentre i multicomputer sono facili da costruire, molte ricerche sono indirizzate alla realizzazione di sistemi ibridi che uniscano le qualità di entrambi. Tali calcolatori cercano di dare l'illusione che esista una memoria condivisa, senza però averne realmente una in quanto troppo costosa. Nel Capitolo 8 analizzeremo in dettaglio multiprocessori e multicomputer.

2.2 Memoria principale

La **memoria** è quella parte del calcolatore in cui sono depositati programmi e dati. Alcuni informatici (specialmente quelli britannici) utilizzano il termine inglese ***store*** o ***storage*** ("immagazzinare", "immagazzinamento") al posto di memoria, anche se il termine *storage* viene utilizzato sempre più frequentemente per riferirsi alla registrazione dei dati nei dischi. Se non ci fosse una memoria da cui il processore potesse leggere e scrivere informazioni, non esisterebbero i calcolatori digitali a programma memorizzato.

2.2.1 Bit

L'unità base della memoria è la cifra binaria, chiamata **bit**. Un bit può avere valore 0 oppure 1 ed è l'unità più semplice possibile. (Difficilmente un dispositivo in grado di memorizzare soltanto 0 potrebbe essere alla base di un sistema di memorizzazione; è infatti necessario che ci siano almeno due valori distinti.)

Quando si dice che i calcolatori utilizzano l'aritmetica binaria perché è "efficiente", s'intende (anche se spesso non ci si rende conto) che l'informazione digitale può essere memorizzata utilizzando dei valori di una certa quantità fisica continua, come la tensione o la corrente. Se occorre distinguere più valori, allora ci deve essere una minor separazione tra valori adiacenti e la memoria risulta di conseguenza meno affidabile. Dato che la numerazione binaria (Appendice A) richiede due soli valori distinti, risulta il metodo più affidabile per codificare l'informazione digitale.

Alcuni calcolatori, tra i quali i mainframe IBM, sono pubblicizzati affermando che sono dotati di aritmetica digitale oltre di quella binaria. Il trucco utilizzato consiste nell'usare 4 bit per memorizzare una cifra decimale mediante un codice chiamato **BCD** (*Binary Coded Decimal*, "decimale codificato in binario"). Quattro bit forniscono 16 combinazioni, di cui 10 sono utilizzate per le cifre da 0 a 9, mentre 6 sono inutilizzate. Di seguito mostriamo la codifica decimale e binaria del numero 1944, utilizzando 16 bit in entrambi i casi:

```
decimale: 0001 1001 0100 0100          binaria: 0000011110011000
```

Nel formato decimale sedici bit possono memorizzare i numeri da 0 a 9999, permettendo solo 10.000 combinazioni, mentre una stringa binaria di 16 bit può avere 65.536 combinazioni distinte. Per questa ragione si dice che il sistema binario è più efficiente.

Consideriamo tuttavia che cosa accadrebbe se un giovane e brillante ingegnere elettronico inventasse un dispositivo altamente affidabile in grado di memorizzare direttamente le cifre da 0 a 9 dividendo in 10 intervalli la regione compresa tra 0 e 10 Volt. Quattro di questi dispositivi potrebbero memorizzare un qualsiasi numero decimale tra 0 e 9999, fornendo così 10.000 combinazioni. Essi potrebbero essere utilizzati anche per memorizzare numeri binari usando solamente le cifre 0 e 1, ma in questo caso, con quattro dispositivi, si otterrebbero soltanto 16 combinazioni. Con un simile dispositivo il sistema decimale risulterebbe ovviamente più efficiente.

2.2.2 Indirizzi di memoria

Le memorie sono costituite da un certo numero di **celle** (o **locazioni**) ciascuna delle quali può memorizzare informazioni. Ciascuna cella ha un numero, chiamato **indirizzo**, attraverso il quale il programma può riferirsi a essa. Se una memoria ha n celle, i suoi indirizzi varieranno da 0 a $n-1$. Tutte le celle di una memoria contengono lo stesso numero di bit; se una cella è costituita da k bit, essa può contenere una qualsiasi delle 2^k diverse combinazioni di bit. La Figura 2.9 mostra tre diverse organizzazioni di una memoria a 96 bit; si noti che (per definizione) le celle adiacenti hanno indirizzi consecutivi.

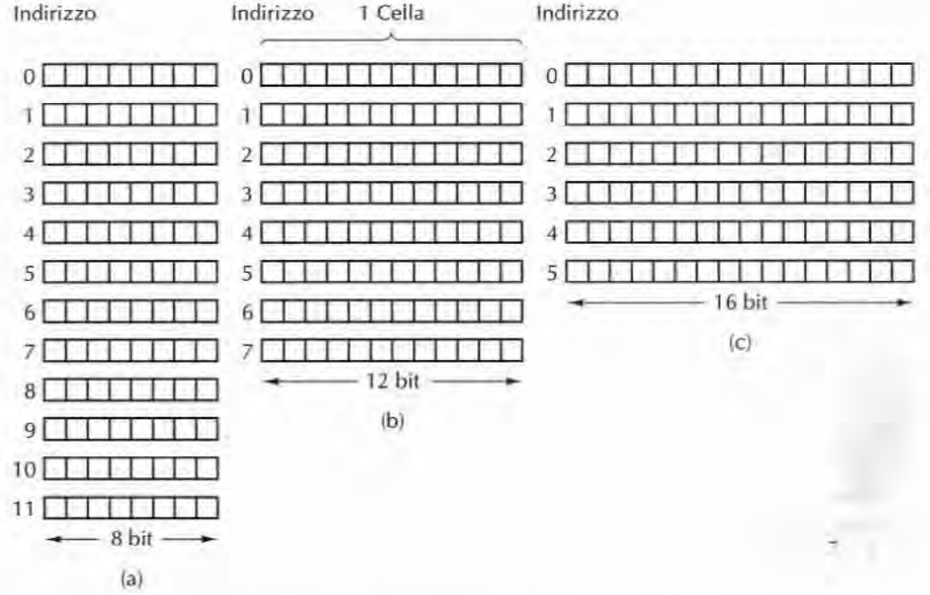


Figura 2.9 Tre modi di organizzare una memoria a 96 bit.

I calcolatori che usano il sistema numerico binario (comprese la notazione ottale ed esadecimale) esprimono gli indirizzi di memoria in notazione binaria. Se un indirizzo ha m bit, il massimo numero di celle indirizzabili è 2^m. Per esempio un indirizzo usato per referenziare la memoria della Figura 2.9(a) richiede almeno 4 bit per poter esprimere tutti i numeri da 0 a 11. Per la Figura 2.9(b) e (c) è sufficiente invece un indirizzo a 3 bit. Il numero di bit nell'indirizzo determina il massimo numero di celle di memoria indirizzabili direttamente ed è indipendente dal numero di bit delle celle; sia una memoria con 2^{12} celle di 8 bit sia una memoria con 2^{12} celle di 64 bit richiedono infatti indirizzi a 12 bit.

La Figura 2.10 elenca il numero di bit per cella per alcuni calcolatori commerciali.

Calcolatore	Bit
Burroughs B1700	1
PC IBM	8
DEC PDP-8	12
IBM 1130	16
DEC PDP-15	18
XDS 940	24
Electrologica X8	27
XDS Sigma 9	32
Honeywell 6180	36
CDC 3600	48
CDC Cyber	60

Figura 2.10 Numero di bit per cella per alcuni calcolatori commerciali d'importanza storica.

La cella rappresenta la più piccola unità indirizzabile; negli ultimi anni quasi tutti i produttori di calcolatori ne hanno standardizzato la dimensione impostandola a 8 bit. Questa dimensione è chiamata **byte**; i byte sono raggruppati in **parole** (*word*). Un calcolatore con parole a 32 bit ha 4 byte per parola, mentre un calcolatore con parole a 64 bit ha 8 byte per parola. L'importanza della parola risiede nel fatto che la maggior parte delle istruzioni operano su intere parole, sommandone per esempio due fra loro. Una macchina a 32 bit avrà quindi registri a 32 bit e istruzioni per manipolare parole a 32 bit, mentre una macchina a 64 bit avrà registri a 64 bit e istruzioni per spostare, sommare, sottrarre e manipolare parole a 64 bit.

2.2.3 Ordinamento dei byte

All'interno di una parola i byte possono essere numerati da sinistra a destra oppure da destra a sinistra. Di primo acchito questa scelta potrebbe sembrare insignificante, ma, come mostreremo tra poco, essa presenta degli importanti risvolti. La Figura 2.11(a) rappresenta una parte della memoria di un calcolatore a 32 bit i cui byte sono numerati da sinistra a destra, come nel caso dello SPARC o dei mainframe IBM. La Figura 2.11(b) fornisce un'analoga rappresentazione per un calcolatore a 32 bit che usa la numerazione da destra a sinistra, come la famiglia Intel. Il primo sistema, in cui la numerazione comincia a partire dall'estremo più "grande" (cioè dal byte più significativo) è chiamato *big endian*, in contrapposizione con il sistema *little endian* della Figura 2.11(b). Questi termini si devono a Jonathan Swift che, nel suo libro *I viaggi di Gulliver*, satireggiava i politici che facevano la guerra per la disputa sul modo in cui rompere le uova sode: a partire dalla punta stretta (*little end*) o da quella più larga (*big end*)? Nel campo dell'architettura degli elaboratori il termine è stato utilizzato per la prima volta in un articolo di Cohen del 1981.

È importante capire che sia nei sistemi *big endian* sia in quelli *little endian* un intero a 32 bit, per esempio con valore numerico 6, è rappresentato dai bit 110 nei 3 bit più a destra (meno significativi) di una parola e zero nei 29 bit più a sinistra. Nello schema di tipo

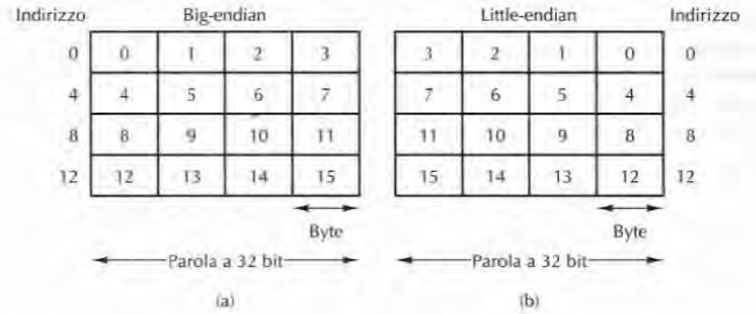


Figura 2.11 (a) Memoria *big endian*. (b) Memoria *little endian*.

big endian i bit 110 si trovano nel byte 3 (o 7, o 11, ecc.), mentre in quello di tipo *little endian* nel byte 0 (o 4, o 8, ecc.). In entrambi i casi la parola che contiene questo indirizzo ha indirizzo 0.

Se i calcolatori memorizzassero soltanto gli interi non ci sarebbe nessun problema. Tuttavia molte applicazioni richiedono interi, stringhe di caratteri e altri tipi di dati. Consideriamo per esempio un semplice archivio per il personale con record costituiti da una stringa (il nome dell'impiegato) e due interi (l'età e il numero del reparto). La stringa termina con uno o più byte con valore 0 in modo da riempire interamente una parola. La Figura 2.12(a) mostra la rappresentazione *big endian* di John Smith, età 21 anni, dipartimento 260 (1 × 256 + 4 = 260); la rappresentazione *little endian* è data invece nella Figura 2.12(b).

Le due rappresentazioni sono corrette e coerenti internamente. I problemi nascono nel momento in cui una di queste due macchine prova a spedire via rete il record all'altra macchina. Assumiamo che la macchina *big endian* spedisca il record a quella *little endian*, un byte alla volta, partendo dal byte 0 e terminando con il byte 19 (inoltre, con un certo otti-

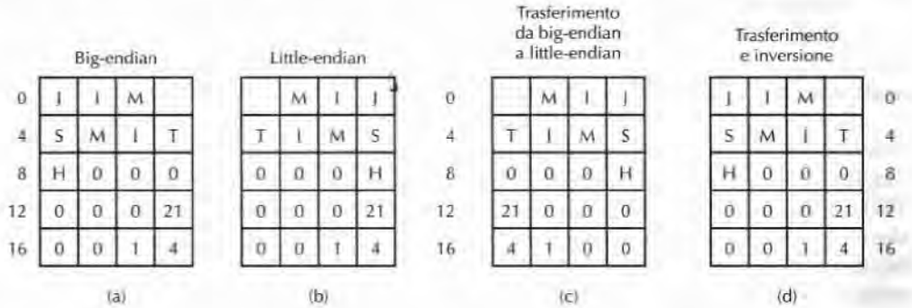


Figura 2.12 (a) Un record dell'archivio del personale per una macchina *big endian*. (b) Lo stesso elemento per una macchina *little endian*. (c) Il risultato del trasferimento dell'elemento da una macchina *big endian* a una *little endian*. (d) Il risultato dell'inversione dei byte di (c).

mismo, assumeremo che non vi siano bit e byte riservati per la trasmissione). Di conseguenza, com'è mostrato nella Figura 2.12(c), il byte 0 del calcolatore *big endian* va a finire al byte 0 della memoria della macchina *little endian*, e così via.

Quando il sistema *little endian* prova a stampare il nome, tutto funziona correttamente, ma l'età viene trasformata in 21×2^{24} e anche il numero del dipartimento assume un valore diverso. Questa situazione si verifica perché la trasmissione ha invertito, com'è giusto che sia, l'ordine dei caratteri all'interno una parola, ma ha allo stesso tempo invertito anche i byte all'interno di un intero, cosa che invece non dovrebbe accadere.

Una prima soluzione potrebbe essere quella di usare un programma che inverte i byte all'interno di una parola dopo che la copia è stata effettuata. Facendo in questo modo si otterrebbe quanto mostrato nella Figura 2.12(d) dove i due interi sono corretti, mentre la stringa viene trasformata in "MIJTIMS" con la lettera "H" che si perde nel vuoto. L'inversione della stringa si verifica perché il calcolatore, quando la legge, interpreta inizialmente il byte 0 (uno spazio), poi il byte 1 (M), e così via.

Non esiste una soluzione semplice al problema. Un metodo, ben poco efficiente, è quello di includere un'intestazione prima di ogni dato per specificarne il tipo (stringa, intero o altro) e la lunghezza, in modo che il ricevente possa effettuare solo le conversioni realmente necessarie. In ogni caso dovrebbe essere chiaro che la mancanza di uno standard per l'ordinamento dei byte costituisce una grande scomodità nello scambio di dati fra macchine di tipo diverso.

2.2.4 Codici correttori

Occasionalmente le memorie dei calcolatori possono commettere degli errori per via di picchi di tensione sulle linee di alimentazione o per altre cause. Per proteggersi da tali errori, alcune memorie utilizzano dei codici di rilevazione e/o di correzione degli errori. Quando si impiegano questi codici vengono aggiunti dei bit extra a ogni parola di memoria secondo una modalità particolare. Quando una parola viene letta dalla memoria, si controllano questi bit aggiuntivi per vedere se si è verificato un errore.

Per capire come sia possibile gestire gli errori, è necessario comprendere in dettaglio che cosa si intende realmente per "errore". Supponiamo che una parola di memoria consista di m bit di dati ai quali aggiungiamo r bit ridondanti, o di controllo; sia quindi $n = m + r$ la lunghezza totale. Un'unità di n bit contenente m bit di dati e r bit di controllo è spesso chiamata **parola di codice** (*codeword*) a n bit.

Date due parole di codice, diciamo 10001001 e 10110001, è possibile determinare il numero di bit corrispondenti rispetto ai quali differiscono. In questo caso la differenza è di 3 bit. Per determinare il numero di bit diversi, bisogna semplicemente calcolare l'OR ESCLUSIVO (XOR) tra i bit delle due parole e contare quanti bit valgono 1 nel risultato; questo valore è chiamato **distanza di Hamming** (Hamming, 1950). Il suo significato principale è che, se fra due parole di codice vi è una distanza di Hamming pari a d, allora saranno necessari d errori singoli per trasformare una parola nell'altra. Per esempio le parole di codice 11110001 e 00110000 hanno distanza di Hamming 3 in quanto servono 3 errori singoli per convertire una parola nell'altra.

Con una parola di memoria a m bit tutte le 2^m combinazioni di bit sono legali, ma, per via del modo in cui sono calcolati i bit di controllo, solo 2^m delle 2^n parole di codice sono valide. Se una lettura da memoria restituisce una parola di codice non valida, il calcolatore sa che è avvenuto un errore di memoria. Conoscendo l'algoritmo che calcola i bit di controllo è possibile costruire una lista completa delle parole di codice lecite e, da questa lista, trovare le due parole di codice la cui distanza di Hamming è minima; questa distanza è la distanza di Hamming dell'intero codice.

Le proprietà di rilevazione e di correzione degli errori di una parola di codice dipendono dalla sua distanza di Hamming. Per rilevare d errori singoli è necessaria una parola con distanza $d + 1$, poiché con tale codice non esiste alcun modo in cui d errori singoli possano cambiare una parola valida in un'altra parola di codice valida. In modo analogo per correggere d errori singoli è necessario un codice con distanza $2d + 1$, poiché in questo modo le parole di codice legali sono sufficientemente lontane l'una dall'altra; anche con d cambiamenti la parola di codice originaria continua infatti a essere più vicina rispetto a tutte le altre parole di codice, potendola quindi determinare in modo univoco.

Come semplice esempio di codice a correzione di errore consideriamo un codice nel quale al dato si aggiunge un singolo **bit di parità**, scelto in modo che il numero di bit 1 nella parola di codice sia pari (oppure dispari). Un codice di questo tipo ha distanza 2, dato che ogni errore singolo genera una parola di codice la cui parità è errata. In altre parole servono due errori singoli per passare da una parola di codice valida a un'altra anch'essa valida. Questo codice può quindi essere utilizzato per rilevare errori singoli; ogniqualvolta viene letta dalla memoria una parola la cui parità è errata viene segnalata una condizione di errore. Il programma non può continuare, ma almeno non si calcolano alcun risultati errati.

Come semplice esempio consideriamo un codice con solo quattro parole di codice valide:

`0000000000, 0000011111, 1111100000 e 1111111111`

Questo codice ha una distanza 5, il che significa che può correggere errori doppi. Se arriva la parola di codice 0000000111 il ricevente sa che l'originale doveva essere 0000011111 (se gli errori verificatisi erano al massimo doppi). Tuttavia un errore triplo che modifica per esempio la parola 0000000000 nella parola 0000000111 non può essere corretto.

Immaginiamo di voler progettare un codice con m bit di dati e r bit di controllo, capace di correggere tutti gli errori singoli. Ciascuna delle 2^m parole di memoria legali ha n parole di codice illegali a distanza 1 da essa. Queste sono formate invertendo sistematicamente ciascuno degli n bit nella parola di codice generata dalla parola di memoria considerata. Ciascuna delle 2^m parole di memoria legali richiede quindi $n + 1$ stringhe di bit a essa dedicate (n per i possibili errori e 1 per la combinazione corretta). Dato che il numero totale di combinazioni di bit è 2^n deve valere $(n + 1)2^m \leq 2^n$; questa disequazione può essere riscritta come $(m + r + 1) \leq 2^r$, dato che $n = m + r$. Conoscendo m, essa fornisce un limite inferiore al numero di bit di controllo richiesti per correggere errori singoli. La Figura 2.13 mostra il numero di bit di controllo richiesti per parole di memoria di varie lunghezze.

È possibile ottenere questo limite teorico utilizzando un metodo ideato da Richard Hamming. Prima di analizzare l'algoritmo di Hamming, guardiamo una semplice rappresentazione grafica che illustra chiaramente l'idea di un codice a correzione di errore per parole a 4 bit. Il diagramma di Venn della Figura 2.14(a) contiene tre cerchi, A, B e C, che insieme formano sette regioni. Come esempio codifichiamo la parola di memoria a 4 bit 1100 nelle regioni AB, ABC, AC e BC, inserendo un bit in ogni regione (in ordine alfabetico). Questa codifica è mostrata nella Figura 2.14(a).

Dimensione parola	Bit di controllo	Dimensione totale	Percentuale di overhead
8	4	12	50
16	5	21	31
32	6	38	19
64	7	71	11
128	8	136	6
256	9	265	4
512	10	522	2

Figura 2.13 Numero di bit di controllo per un codice che può correggere errori singoli.

Aggiungiamo poi un bit di parità a ciascuna delle tre regioni vuote per ottenere al loro interno una parità pari, come illustrato nella Figura 2.14(b). Per costruzione la somma dei bit in ciascuno dei tre cerchi, *A*, *B* e *C*, è ora un numero pari. Nel cerchio *A* abbiamo i quattro numeri 0, 0, 1 e 1, la cui somma fa 2, un numero pari. Nel cerchio *B* i numeri sono 1, 1, 0 e 0, che sommati fanno ancora 2, un numero pari. Infine nel cerchio *C* abbiamo nuovamente la stessa situazione. In questo esempio tutti i cerchi hanno la stessa somma, ma, in altre situazioni, è possibile ottenere somme diverse, come 0 e 4. Questa figura corrisponde a una parola di codice con 4 bit di dati e 3 bit di parità.

Supponiamo ora che il bit nella regione *AC* diventi errato, passando dal valore 0 al valore 1, come mostrato nella Figura 2.14(c). A questo punto il calcolatore può vedere che i cerchi *A* e *C* hanno la parità errata (dispari). L'unico cambiamento di un singolo bit che li può correggere corrisponde a riportare *AC* al valore 0; questa modifica corregge correttamente l'errore. Seguendo questa strategia il calcolatore può riparare automaticamente gli errori di memoria che coinvolgono un solo bit.

Vediamo ora come l'algoritmo di Hamming può essere usato per costruire codici a correzione di errore per parole di memoria dalla dimensione arbitraria. In un codice di Hamming gli r bit di parità sono aggiunti a una parola a m bit, formando una nuova parola di lunghezza $m + r$ bit. I bit sono numerati a partire da 1, non 0, con il bit 1 nella posizione più a sinistra (più significativa). Tutti i bit la cui posizione è una potenza di 2 sono bit di parità;

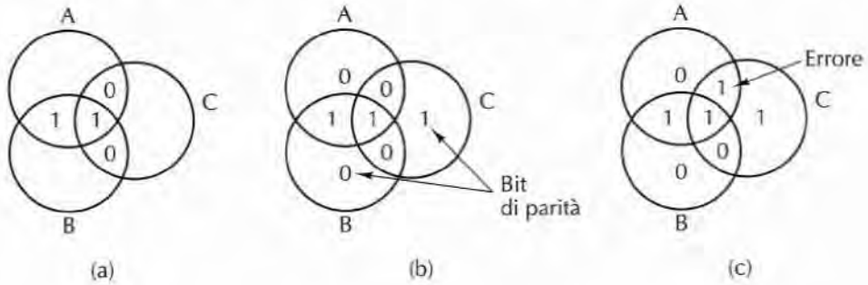


Figura 2.14 (a) Codifica di 1100. (b) Aggiunta della parità pari. (c) Errore in *AC*.

quelli restanti sono usati invece per i dati. Per esempio, con una parola a 16 bit, vengono aggiunti 5 bit di parità nelle posizioni 1, 2, 4, 8 e 16, mentre le altre contengono bit di dati. In totale la parola di memoria diventa di 21 bit (16 di dati, 5 di parità). In questo esempio useremo (arbitrariamente) la parità pari.

Ciascun bit di parità controlla alcune posizioni dei bit specifiche ed è impostato in modo che sia pari il numero totale di bit che hanno valore 1 nelle posizioni controllate. Ogni bit di parità controlla le seguenti posizioni:

```
il bit 1 controlla i bit 1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21
il bit 2 controlla i bit 2, 3, 6, 7, 10, 11, 14, 15, 18, 19
il bit 4 controlla i bit 4, 5, 6, 7, 12, 13, 14, 15, 20, 21
il bit 8 controlla i bit 8, 9, 10, 11, 12, 13, 14, 15
il bit 16 controlla i bit 16, 17, 18, 19, 20, 21.
```

In generale il bit di posto b è controllato dai bit b_1, b_2, ..., b_j tali che $b_1 + b_2 + \ldots + b_j = b$. Per esempio il bit 5 è controllato dai bit 1 e 4 dato che 1 + 4 = 5. Il bit 6 è controllato dai bit 2 e 4 in quanto 2 + 4 = 6, e così via.

La Figura 2.15 mostra la correzione di un codice di Hamming per la parola di memoria a 16 bit 1111000010101110. La parola di codice a 21 bit corrispondente è 00101111000001011101110. Per vedere come funziona la correzione di errore consideriamo che cosa accadrebbe se il bit 5 fosse invertito da un sovraccarico di tensione sulla linea di alimentazione. La nuova parola di codice diventerebbe 00100111000001011101110 (invece di 00101111000001011101110). Controllando i 5 bit di parità si otterrebbero i seguenti risultati:

```
bit di parità 1 non corretto (1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21
   contengono cinque 1)
bit di parità 2 corretto (2, 3, 6, 7, 10, 11, 14, 15, 18, 19
   contengono sei 1)
bit di parità 4 non corretto (4, 5, 6, 7, 12, 13, 14, 15, 20, 21
   contengono cinque 1)
bit di parità 8 corretto (8, 9, 10, 11, 12, 13, 14, 15 contengono due 1)
bit di parità 16 corretto (16, 17, 18, 19, 20, 21 contengono quattro 1).
```

Il numero totale di 1 nelle posizioni 1, 3, 5, 7, 9, 11, 13, 15, 17, 19 e 21 dovrebbe essere un numero pari dato che si sta usando la parità pari. Il bit non corretto deve essere uno dei bit controllati dal bit di parità 1, cioè uno fra i bit 1, 3, 5, 7, 9, 11, 13, 15, 17, 19 e 21. Il

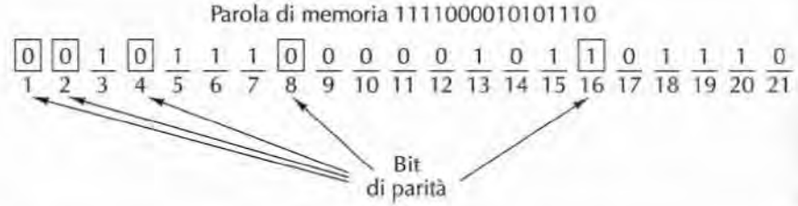


Figura 2.15 Costruzione del codice di Hamming per la parola di memoria 1111000010101110 aggiungendo 5 bit di controllo ai 16 bit di dati.

bit di parità 4 è errato, il che significa che uno fra i bit 4, 5, 6, 7, 12, 13, 14, 15, 20 e 21 non è corretto. L'errore deve essere uno dei bit presenti in entrambe le liste, cioè il bit 5, 7, 13, 15 oppure 21. Tuttavia il bit 2 è errato e quindi i bit 7 e 15 sono eliminati. Analogamente il bit 8 è corretto, eliminando così il bit 13. Infine, dato che anche il bit 16 è corretto, va eliminato pure il 21. Il solo bit rimasto è il bit 5, che è quindi quello in cui si è verificato l'errore; dato che è stato letto come 1, esso deve assumere il valore 0. Questa successione di considerazioni consente di correggere gli errori.

Un semplice metodo per trovare i bit errati consiste nel calcolare inizialmente tutti i bit di parità. Se sono tutti corretti allora non si è verificato alcun errore (oppure più di uno). Successivamente si sommano tutti i bit di parità errati, contando 1 per il bit 1, 2 per il bit 2, 4 per il bit 4, e così via, e la somma risultante corrisponde alla posizione del bit errato. Se per esempio i bit di parità 1 e 4 sono errati, ma 2, 8 e 16 sono corretti, significa che il bit 5 (1 + 4) è stato invertito.

2.2.5 Memoria cache

Storicamente le CPU sono sempre state più veloci delle memorie; i miglioramenti di quest'ultime hanno contribuito a incremetare anche le prestazioni delle CPU, preservando di fatto lo squilibrio. Infatti, dato che diventa possibile collocare sempre più circuiti su un chip, i progettisti delle CPU stanno usando queste nuove possibilità per sviluppare le architetture a pipeline e superscalari, migliorando ulteriormente le velocità delle CPU. Al contrario i progettisti delle memorie hanno generalmente usato questa nuova tecnologia per aumentare la capacità dei loro chip, ma non la loro velocità, facendo peggiorare con il tempo lo squilibrio tra i due componenti. Il significato pratico di questa differenza di prestazioni è che, quando la CPU lancia una richiesta alla memoria, essa non otterrà la parola desiderata se non dopo molti cicli di CPU. Più lenta è la memoria, più cicli dovrà attendere la CPU.

Come abbiamo precisato precedentemente esistono due modi per trattare questo problema. Il metodo più semplice consiste nel far iniziare le istruzioni di lettura dalla memoria non appena vengono incontrate, permettendo al contempo di continuare l'esecuzione e bloccando la CPU quando un'istruzione tenta di usare una parola di memoria non ancora arrivata. Più lenta è la memoria, più frequentemente si verifica questo problema e maggiore è la penalizzazione che ne consegue. Se per esempio il ritardo della memoria è di 10 cicli è molto probabile che una delle 10 istruzioni successive tenti di utilizzare la parola letta.

L'altra soluzione consiste nel richiedere ai compilatori di non generare codice che utilizzi parole ancor prima che queste siano arrivate, il che consente di avere macchine che non si bloccano. Il problema di questo approccio è che è molto più facile a dirsi che a farsi. Spesso dopo una `LOAD` non vi è nient'altro da fare e quindi il compilatore è forzato a inserire delle istruzioni fasulle, dette `NOP` (cioè nessuna operazione), che non eseguono alcuna azione, ma occupano uno slot di tempo. In sostanza questo approccio è uno stallo software invece che hardware; il peggioramento delle prestazioni non cambia.

Attualmente il problema non sta nella tecnologia, ma piuttosto in considerazioni economiche. Gli ingegneri sanno come costruire memorie veloci quanto le CPU, ma, per poter andare alla stessa velocità, esse devono essere collocate sul chip della CPU (dato che utilizzare il bus per la memoria è troppo lento). Inserire una grande memoria sul chip della CPU la rende più grande, il che significa più costosa; inoltre, anche se il costo non fosse un problema, esistono comunque dei limiti alla dimensione di un chip di CPU. La scelta si riduce quindi tra avere una piccola quantità di memoria veloce o una grande quantità di memoria più lenta, anche se, idealmente, vorremmo avere una grande quantità di memoria veloce a un prezzo basso.

Nonostante queste limitazioni, ci sono delle tecniche interessanti che permettono di combinare una piccola quantità di memoria veloce con una grande quantità di memoria lenta al fine di ottenere a un prezzo modesto sia la velocità della memoria veloce (quasi) sia la capacità della memoria più grande. La piccola e veloce memoria è chiamata ***cache*** (dal francese *cacher*, che significa "nascondere")[4]. Ora descriveremo brevemente come si usano queste memorie e come funzionano, mentre una descrizione più dettagliata sarà data nel Capitolo 4.

L'idea di base è semplice: le parole di memoria usate più di frequente sono mantenute all'interno della cache. Quando la CPU necessita di una parola, la cerca nella cache e, solo nel caso in cui essa non sia presente, la richiede alla memoria centrale. È possibile ridurre drasticamente il tempo medio di accesso se una frazione significativa delle parole è presente nella cache.

Il successo o il fallimento dipendono quindi da quali parole sono presenti nella cache. Da anni si sa che i programmi non accedono alle loro memorie in modo completamente casuale. Se ad un certo istante la memoria fa un riferimento all'indirizzo A è molto probabile che il successivo riferimento alla memoria si troverà nelle vicinanze di A. Un semplice esempio è il programma stesso, in quanto, fatta eccezione per i salti e le chiamate a procedura, le istruzioni sono prelevate da locazioni contigue di memoria. Inoltre la maggior parte del tempo di esecuzione di un programma è spesa nei cicli, nei quali si esegue ripetutamente un numero limitato d'istruzioni. Analogamente, è molto probabile che un programma per la manipolazione di matrici effettuerà svariati riferimenti alla stessa matrice prima di spostarsi su altri dati.

L'osservazione secondo la quale i riferimenti alla memoria fatti in un breve intervallo temporale tendono a utilizzare solo una piccola frazione della memoria totale è chiamata **principio di località** ed è alla base di tutti i sistemi di cache. L'idea generale prevede che quando una parola viene referenziata, la parola stessa e alcune parole vicine sono portate dalla grande e lenta memoria all'interno della cache, in modo che sia possibile accedervi velocemente in un secondo momento. La Figura 2.16 mostra una tipica organizzazione di CPU, cache e memoria centrale. Se durante un piccolo intervallo una parola è letta o scritta k volte, il calcolatore dovrà effettuare un solo riferimento alla memoria lenta e $k-1$ riferimenti alla memoria veloce. Maggiore è k, migliori sono le prestazioni complessive.

Possiamo formalizzare questo calcolo introducendo, il tempo di accesso alla cache, il tempo di accesso alla memoria centrale, la frequenza di successi (*hit ratio*), che corrisponde alla frazione di riferimenti che può essere soddisfatta dalla cache. Indicheremo tali grandezze rispettivamente con le lettere c, m, e h. Nel semplice esempio del paragrafo precedente si ottiene $h = (k-1)/k$. Alcuni autori definiscono anche la frequenza di fallimento (*miss ratio*), che vale $1 - h$.

---

[4] In inglese la parola *cache* indica un deposito, una scorta di beni da utilizzare al momento opportuno (*N.d.T.*).

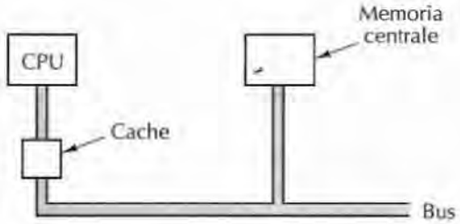


Figura 2.16 Da un punto di vista logico la cache si trova tra la CPU e la memoria centrale. Fisicamente può essere collocata in varie posizioni.

È quindi possibile calcolare il tempo medio di accesso utilizzando la seguente relazione:

$$\text{Tempo medio di accesso} = c + (1 - h)\ m$$

Se h si avvicina a 1, tutti i riferimenti possono essere soddisfatti dalla cache e il tempo medio di accesso si approssima a c. Al contrario, quando h tende a 0 è necessario effettuare ogni volta un riferimento alla memoria e quindi il tempo di accesso medio si avvicina a $c + m$, ovvero alla somma del tempo c necessario per controllare inizialmente (senza successo) la cache e del tempo m per fare successivamente riferimento alla memoria. Su alcuni sistemi il riferimento alla memoria può essere fatto partire parallelamente alla ricerca nella cache, di modo che, se si verifica un fallimento della cache, il ciclo di memoria sia già iniziato. Tuttavia questa strategia richiede che la memoria possa essere bloccata in ogni momento quando si verifica un successo della cache, rendendo l'implementazione più complicata.

Le memorie centrali e le cache sono divise in blocchi di grandezza fissa per trarre vantaggio dal principio di località. Solitamente quando si parla di questi blocchi all'interno della cache, ci si riferisce a essi con il termine **linea di cache**. Quando si verifica un fallimento della cache, è l'intera linea a essere caricata dalla memoria centrale all'interno della cache, e non soltanto la parola richiesta. Per esempio con una linea di 64 byte, un riferimento all'indirizzo di memoria 260 porterà la linea compresa tra i byte 256 e 319 all'interno della linea di cache. Con un po' di fortuna alcune delle altre parole presenti nella linea di cache diventeranno necessarie nel giro di poche istruzioni. È più efficiente seguire questa strategia piuttosto che prelevare singole parole, dato che richiede meno tempo prelevare k parole tutte in una volta che prelevarle una ad una in k momenti diversi. Inoltre il fatto di avere elementi di cache più grandi di una singola parola significa che ce n'è un numero minore, il che richiede un overhead inferiore.

La progettazione delle cache sta acquisendo un'importanza sempre maggiore nelle CPU ad alte prestazioni. Un problema riguarda la grandezza della cache; più grande è, migliori sono le sue prestazioni, ma anche il suo costo è maggiore. Un secondo problema è la dimensione della linea di cache. Una cache da 16 KB può essere infatti divisa in 1024 linee di 16 byte, 2048 linee di 8 byte, o in altri modi. Un terzo problema è come deve essere organizzata la cache, cioè come tener traccia di quali parole di memoria sono memorizzate al suo interno in un dato momento.

Un quarto problema consiste nella scelta se mantenere istruzioni e dati in una sola cache oppure in due. Il fatto di avere una **cache unificata** (istruzioni e dati usano la stessa cache) implica una progettazione più semplice e bilancia automaticamente i prelievi delle istruzioni e dei dati. Al giorno d'oggi tuttavia si tende a favorire la **cache specializzata**, in cui le istruzioni sono memorizzate in una cache e i dati in un'altra. Questa organizzazione è chiamata **architettura Harvard**, in riferimento al vecchio calcolatore Mark III di Howard Aiken, che aveva memorie separate per le istruzioni e per i dati. Ciò che spinge i progettisti in questa direzione è l'utilizzo molto diffuso delle CPU con pipeline in cui l'unità di prelievo dell'istruzione richiede l'accesso alle istruzioni nello stesso momento in cui l'unità di prelievo dell'operando richiede l'accesso ai dati. Le cache specializzate permettono che questi due accessi avvengano in parallelo, mentre con una unificata ciò non è possibile. Inoltre dato che le istruzioni non sono modificate durante l'esecuzione, non vi è mai bisogno di riscrivere in memoria il contenuto della cache delle istruzioni.

Infine un quinto problema è rappresentato dal numero delle cache. Al giorno d'oggi non è raro avere chip con una cache primaria sul chip, una cache secondaria in un altro chip montato in prossimità, e una terza cache ancor più lontana.

2.2.6 Assemblaggio e tipi di memoria

A partire dalle prime memorie basate su semiconduttori e fino ai primi anni '90, la memoria era prodotta, venduta e installata in un unico chip. La densità dei chip passò da 1 Kbit a oltre 1 Mbit, ma ogni chip veniva venduto come unità separata. Spesso i primi PC erano dotati di prese nelle quali era possibile inserire chip di memoria aggiuntivi, se e quando l'utente ne avesse avuto bisogno.

Attualmente si ricorre a una diversa organizzazione. Vari chip, in genere di 8 o 16 elementi, sono montati su una piccola scheda a circuiti stampati venduta singolarmente. Essa è chiamata **SIMM** (*Single Inline Memory Module*, "modulo di memoria con connettore singolo") oppure **DIMM** (*Dual Inline Memory Module*, "modulo di memoria con doppia linea di contatti"), a seconda che abbia i connettori allineati su uno o su due lati della scheda. Le SIMM hanno un connettore laterale con 72 contatti e trasferiscono 32 bit per ciclo di clock; le DIMM invece hanno generalmente connettori con 84 contatti e trasferiscono 64 bit per ciclo di clock. La Figura 2.17 mostra una SIMM schematizzata.

Una tipica configurazione di SIMM o DIMM può avere otto chip di dati con 256 megabit (32 MB) ciascuna, per un totale di 256 MB sull'intero modulo. Molti calcolatori hanno spazio per quattro moduli, permettendo una capacità totale di 1 GB (o anche maggiore).

Nei calcolatori portatili si usano DIMM più piccole, chiamate **SO-DIMM** (*Small Outline DIMM*, "DIMM dal profilo compatto"). Alle SIMM e alle DIMM è possibile ag-

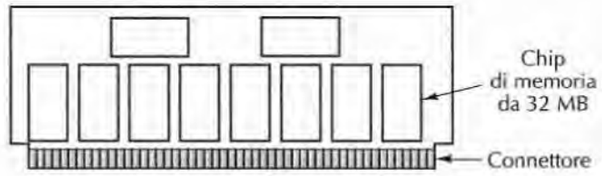


Figura 2.17 Un modulo di memoria con connettore singolo in linea (SIMM) da 256 MB. Due dei chip controllano la SIMM.

giungere un bit di parità o la correzione d'errore, ma, dato che la frequenza media degli errori di un modulo è di uno ogni 10 anni, spesso nella maggior parte dei calcolatori ordinari le funzionalità di rilevazione e correzione degli errori non sono presenti.

2.3 Memoria secondaria

La memoria centrale non è mai troppo grande. Si vorrebbero sempre memorizzare più informazioni rispetto alle reali capacità; la ragione principale è che, grazie ai miglioramenti della tecnologia, si comincia a voler memorizzare cose che precedentemente ricadevano interamente nel regno della fantascienza. Per esempio, visto che il governo degli USA obbliga gli enti governativi a pubblicare i propri bilanci, si potrebbe immaginare che la Biblioteca del Congresso decida di digitalizzare e mettere in commercio tutto il proprio contenuto ("Tutta la conoscenza umana a soli $ 99,95"). Memorizzare all'incirca 50 milioni di libri, ciascuno con 1 MB di testo e 1 MB di immagini compresse, richiede 10^{14} byte, vale a dire 100 Terabyte. La memorizzazione di tutti i 50.000 film finora prodotti richiederebbe più o meno lo stesso spazio. Questa quantità d'informazione, almeno per alcuni decenni, sarà troppo grande per poter essere contenuta in memoria centrale.

2.3.1 Gerarchie di memoria

La soluzione che viene tradizionalmente adottata per memorizzare una gran mole di dati consiste nell'organizzare gerarchicamente la memoria, com'è mostrato nella Figura 2.18. Nella parte alta della gerarchia si trovano i registri della CPU, ai quali si può accedere alla stessa velocità della CPU. Successivamente vi è la memoria cache, la cui dimensione può variare da 32 KB fino ad alcuni megabyte. La memoria centrale è il passo successivo e la sua dimensione è compresa tra 16 MB per i sistemi più economici fino a decine di gigabyte per quelli professionali. Successivamente troviamo i dischi magnetici, la vera forza lavoro per quanto riguarda la memorizzazione permanente. Infine ci sono i nastri magnetici e i dischi ottici utilizzati per l'archiviazione.

Muovendosi verso il basso della gerarchia aumentano tre parametri chiave. Innanzitutto, il tempo di accesso diventa via via più grande. Ai registri della CPU è possibile accedere in pochi nanosecondi. Le memorie cache richiedono invece un tempo che è un piccolo multiplo di quello dei registri della CPU, mentre alle memorie centrali è generalmente possibile accedere in poche decine di nanosecondi. A questo punto si incontra un grande salto, dato che il tempo di accesso ai dischi è di almeno 10 ms e l'accesso ai nastri e ai dischi ottici può essere misurato in secondi se i supporti devono essere presi e inseriti in un lettore.

Secondariamente, la capacità di memorizzazione aumenta man mano si scende verso il basso. I registri della CPU vanno bene per immagazzinare circa 128 byte, mentre le cache pochi megabyte; le memorie centrali possono memorizzare da decine a migliaia di megabyte e i dischi magnetici possono contenere quantità che variano da alcuni gigabyte fino a decine di gigabyte. I nastri e i dischi magnetici sono generalmente scollegati dal sistema e quindi la loro capacità è limitata soltanto dalle disponibilità economiche dell'utente.

In terzo luogo, scendendo lungo la gerarchia diminuiscono anche i costi unitari. Sebbene i prezzi attuali cambino rapidamente, i costi della memoria centrale sono dell'ordine

Figura 2.18 Gerarchia di memoria a cinque livelli.

di euro/megabyte, quelli dei dischi magnetici sono circa 100 volte inferiori e ancora più bassi quelli dei nastri magnetici.

Abbiamo già analizzato registri, cache e memoria centrale; nei paragrafi successivi studieremo i dischi magnetici e i dischi ottici. Non considereremo i nastri magnetici dato che sono usati raramente se non per il backup e, in ogni caso, non vi è molto da dire su di loro.

2.3.2 Dischi magnetici

Un disco magnetico consiste di uno o più piatti di alluminio rivestiti di materiale magnetico. Originariamente il diametro dei piatti poteva raggiungere i 50 cm, mentre al giorno d'oggi variano generalmente tra i 3 e i 12 cm; i dischi per i calcolatori portatili continuano a diventare sempre più piccoli e già oggi le loro dimensioni sono inferiori a 3 cm. La testina del disco, contenente un solenoide, sfiora la superficie rimanendo sospesa su un cuscinetto d'aria (mentre la testina dei floppy disk tocca realmente la superficie). Una corrente che passa attraverso la testina magnetizza la superficie che si trova al di sotto, orientando le particelle magnetiche in direzione opposta a seconda del verso della corrente. Quando la testina passa sopra un'area magnetizzata, viene indotta nella testina una corrente positiva o negativa rendendo così possibile la lettura dei bit memorizzati. In questo modo, mentre il piatto ruota sotto la testina, è possibile scrivere e leggere un flusso di bit. La Figura 2.19 mostra la struttura di una traccia del disco.

La sequenza circolare di bit scritti mentre il disco compie una rotazione completa è chiamata **traccia**. Le tracce sono divise in un certo numero di **settori** di lunghezza fissa, contenenti in genere 512 byte di dati e preceduti da un **preambolo** che permette alla testina di sincronizzarsi prima della lettura o della scrittura. Dopo i dati segue un codice per la correzione di errore (ECC, *Error Correction Code*), un codice di Hamming oppure, più comunemente, un codice capace di correggere errori multipli, chiamato codice **Reed-Solomon**. Tra due settori consecutivi vi è un piccola area chiamata **spazio tra settori**. Alcuni produttori

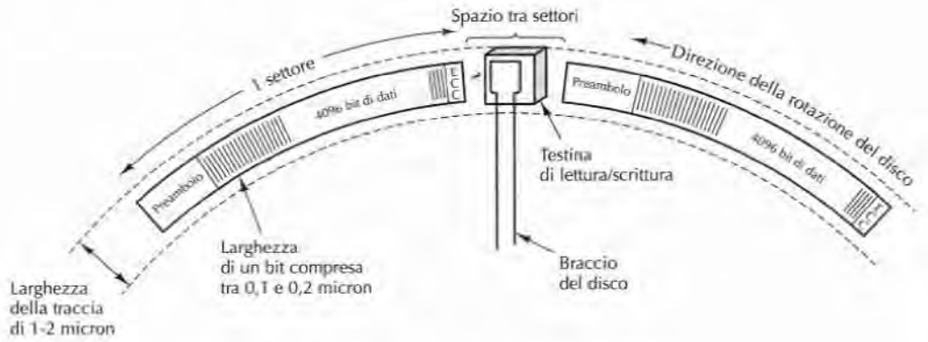


Figura 2.19 Porzione di una traccia del disco. Sono mostrati due settori.

indicano la capacità dei loro dischi nello stato non formattato (come se ciascuna traccia contenesse soltanto dati), ma una misura più onesta dovrebbe essere la capacità formattata, che non conta i preamboli, gli ECC e gli spazi tra i dati. Di solito la capacità formattata è inferiore del 15% circa rispetto a quella lorda.

Tutti i dischi hanno bracci mobili in grado di spostarsi radialmente per posizionarsi su diverse distanze rispetto al centro di rotazione del piatto. Dato che è possibile scrivere una diversa traccia in corrispondenza di ogni diversa distanza radiale, le tracce sono una serie di centri concentrici attorno all'asse di rotazione del disco. La larghezza di una traccia dipende da quanto è larga la testina e da quanto accuratamente può essere posizionata. Con la tecnologia attuale i dischi hanno tra 5000 e 10.000 tracce per centimetro, con una larghezza per traccia che varia tra 1 e 2 μ (1 μ = 1/1000 mm). Occorre sottolineare che una traccia non è una scanalatura fisica nella superficie, ma semplicemente una corona circolare di materiale magnetizzato, con piccole aree di sicurezza che la separano da quella più interna e da quella più esterna.

La densità lineare dei bit lungo le tracce è diversa dalla densità radiale, ed è in gran parte determinata dalla purezza della superficie e dalla qualità dell'aria. I dischi attuali raggiungono densità comprese tra i 50.000 e i 100.000 bit/cm; la dimensione di un bit è quindi circa 50 volte maggiore sia nella direzione radiale sia lungo la circonferenza. Per raggiungere densità ancor più elevate i produttori di dischi stanno sviluppando tecnologie nelle quali la dimensione "lunga" dei bit non è disposta lungo la circonferenza dei dischi, ma verticalmente all'interno del materiale ferromagnetico. Questa tecnica è chiamata **registrazione perpendicolare** e verrà commercializzata a breve.

Al fine di ottenere un'alta qualità della superficie e dell'aria molti dischi sono sigillati durante la fabbricazione per impedire che vi si depositi la polvere. Questi dischi sono chiamati **dischi Winchester** e il primo modello (prodotto da IBM) aveva 30 MB di memoria fissa e sigillata e 30 MB di memoria removibile. Molto probabilmente il nome "Winchester" ha preso piede poiché questi dischi 30-30 fanno tornare alla mente i fucili Winchester 30-30 che ebbero un ruolo importante nell'avanzamento della frontiera americana.

La maggior parte dei dischi consiste di più piatti impilati verticalmente (come rappresentato nella Figura 2.20) in cui ciascuna superficie ha il proprio braccio e la propria

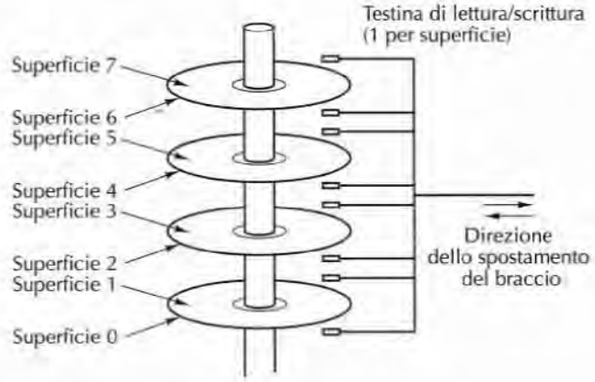


Figura 2.20 Disco con quattro piatti.

testina. L'insieme di tracce alla stessa distanza dal centro è chiamato **cilindro**. I dischi dei PC attuali hanno dai 6 ai 12 piatti, mettendo quindi a disposizione dalle 12 alle 24 superfici registrabili.

Le prestazioni dei dischi dipendono da un insieme di fattori. Per leggere o scrivere un settore il braccio deve inizialmente effettuare quella che viene chiamata **seek** (ricerca), cioè lo spostamento radiale sulla posizione corretta. Il tempo medio di ricerca è compreso tra i 5 e i 10 ms, mentre gli spostamenti tra due tracce consecutive sono attualmente al di sotto del millisecondo. Una volta che la testina si è posizionata radialmente c'è un ritardo, chiamato **latenza rotazionale**, nell'attesa che il settore desiderato ruoti sotto la testina. Dato che la maggior parte dei dischi ruota a 5400, 7200 o 10.800 giri al minuto, il ritardo medio (corrispondente a metà rotazione) è compreso tra i 3 e i 6 ms. Il tempo di trasferimento dipende invece dalla densità lineare e dalla velocità rotazionale. Considerando che le velocità di trasferimento sono comunemente comprese tra i 20 e i 40 MB/s, un settore di 512 byte richiede tra i 13 e i 16 μs; da ciò ne consegue che il tempo di ricerca su disco e la latenza rotazionale dominano la velocità di trasferimento. Leggere casualmente settori sparsi sul disco è dunque un modo chiaramente inefficiente di operare.

Vale la pena sottolineare che, a causa di preamboli, ECC, spazi tra i settori, tempi di ricerca su disco e latenze rotazionali, esiste una gran differenza tra *burst rate* e *sustained rate* di un disco. Il primo è la velocità a cui la testina può leggere dati dopo essersi posizionata sul primo bit. Il calcolatore deve essere in grado di gestire i dati letti a questa velocità (*burst* in inglese significa "raffica"); d'altra parte il disco può mantenere questa velocità soltanto per un settore. Per alcune applicazioni, come quelle multimediali, è più importante il valore del *sustained rate*, cioè la velocità di lettura media nell'arco di un periodo di alcuni secondi, nel quale sono compresi anche i necessari tempi di ricerca e le latenze rotazionali.

Una semplice riflessione e l'uso della formuletta dei tempi del liceo, $c = 2\pi r$, svelano che le tracce più esterne hanno fra loro una distanza lineare maggiore rispetto a quanta non ne abbiano quelle più interne. Dato che, indipendentemente da dove si trova la testina, tutti i dischi magnetici ruotano a velocità angolare costante, questa osservazione mette in evidenza

un problema. Nei dischi più vecchi i produttori erano abituati a massimizzare la densità lineare nelle tracce più interne e ad abbassare la densità lineare di bit su quelle più esterne. Se un disco aveva, per esempio, 18 settori per traccia, allora ognuno di loro occupava un arco di 20 gradi, indipendentemente dal cilindro nel quale si trovasse.

Al giorno d'oggi si utilizza una strategia diversa. I cilindri sono divisi in zone (in genere da 10 a 30) e il numero di settori per traccia aumenta in ciascuna zona man mano che ci si sposta dalla traccia più interna verso l'esterno. Tutti i settori sono della stessa dimensione, ma la loro particolare disposizione rende più difficile tener traccia di dove si trovano le informazioni. Tuttavia questa tecnica permette di aumentare la capacità del disco e ciò è considerato un aspetto di maggiore importanza. La Figura 2.21 mostra un disco con cinque zone.

A ogni disco è associato un processore dedicato, chiamato **controllore del disco**. Fra i compiti del controllore c'è quello di accettare i comandi dal software, come `READ`, `WRITE` e `FORMAT` (scrittura dei preamboli), di controllare il movimento del braccio, di rilevare e correggere gli errori e di convertire gli 8 bit dei byte letti dalla memoria in uno stream seriale di bit e viceversa. Alcuni controllori gestiscono anche la bufferizzazione di più settori, memorizzando in una cache i settori letti per un loro eventuale riutilizzo, oltre alla gestione dei settori contenenti errori. Quest'ultima funzione è necessaria in quanto esistono settori con regioni magnetizzate in modo permanente; quando il controllore scopre uno di questi settori lo sostituisce con uno dei settori liberi appositamente riservati a questo scopo all'interno di ciascun cilindro o di ciascuna zona.

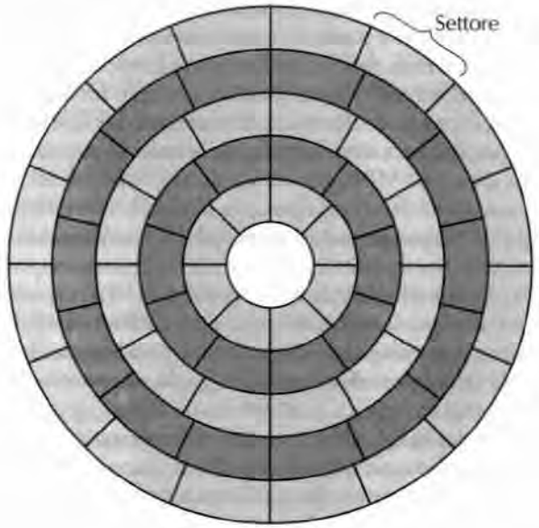


Figura 2.21 Disco con cinque zone. Ciascuna zona ha più tracce.

2.3.3 Floppy disk

Con l'avvento del personal computer nacque la necessità di trovare un modo di distribuire il software. La soluzione fu trovata nel **dischetto** o **floppy disk**, un piccolo supporto rimovibile così chiamato in quanto le prime versioni erano fisicamente flessibili. Il floppy disk fu in realtà inventato dall'IBM per un altro scopo: quello di registrare le informazioni di manutenzione dei propri mainframe per il personale di servizio. In breve tempo però i produttori di personal computer cominciarono a utilizzarlo come un comodo mezzo per distribuire software a pagamento.

Le caratteristiche generali sono le stesse dei dischi descritti precedentemente, tranne per il fatto che, diversamente dagli hard disk, le testine dei floppy disk toccano realmente i dischetti. Per questo motivo sia i supporti sia le testine si consumano in modo relativamente veloce. Quando l'unità non sta né leggendo né scrivendo dati i personal computer ritirano le testine e arrestano la rotazione per ridurre il deterioramento fisico; di conseguenza quando giunge un nuovo comando di lettura o scrittura si verifica un ritardo di circa mezzo secondo prima che il motore raggiunga la velocità corretta. I floppy disk hanno una durata di circa 20 anni, ma ormai non vengono quasi più venduti insieme ai moderni calcolatori.

2.3.4 Dischi IDE

I dischi dei moderni personal computer sono un'evoluzione di quello che era presente nel PC XT della IBM; un disco da 10 MB della Seagate gestito da un controllore prodotto dalla Xebec collocato su una scheda aggiuntiva. Il disco della Seagate aveva 4 testine, 306 cilindri e 17 settori per traccia e il controllore era in grado di gestire due unità. Il sistema operativo leggeva dal disco e scriveva su di esso inserendo alcuni parametri nei registri della CPU e chiamando successivamente il **BIOS** (*Basic Input Output System*, "sistema elementare di input/output"), situato in una memoria di sola lettura integrata nel PC. Il BIOS lanciava le istruzioni macchina necessarie per caricare i registri del controllore, che iniziava il trasferimento dei dati.

La tecnologia evolse rapidamente e a metà degli anni '80 si passò alle prime unità **IDE** (*Integrated Drive Electronics*, "memoria di massa con elettronica integrata") in cui il controllo era strettamente integrato con l'unità, mentre prima si trovava su una scheda separata. Tuttavia, per ragioni di retrocompatibilità, le chiamate al BIOS non furono modificate. Secondo queste convenzioni l'indirizzamento dei settori avveniva specificando i numeri della loro testina, del cilindro e del settore, con la numerazione delle testine e dei cilindri che partiva da 0, mentre quella dei settori partiva da 1. Probabilmente questa scelta fu dovuta a un errore da parte del programmatore del BIOS originario, che scrisse il suo capolavoro usando l'assemblatore dell'8088. Utilizzando 4 bit per la testina, 6 bit per il settore e 10 bit per il cilindro, un'unità poteva avere al massimo 16 testine, 63 settori e 1024 cilindri, per un totale di 1.032.192 settori. Una simile unità poteva avere una capacità massima di 504 MB, che all'epoca poteva sembrare praticamente infinita, ma che al giorno d'oggi non lo è certamente. (Si criticherebbe oggi una macchina che non sia in grado di gestire dei dischi più grandi di un petabyte[5]?)

---

[5] 1 peta = 10^{15} (*N.d.Rev.*).

Successivamente apparvero unità più capaci, la cui struttura (4 testine, 32 settori, 2000 cilindri) non poteva essere sfruttata; a causa delle convenzioni sulle chiamate al BIOS, rimaste per lungo tempo immutate, il sistema operativo non poteva infatti indirizzare questi dischi. Per aggirare il problema i controllori dei dischi cominciarono a imbrogliare, richiedendo che la geometria fosse all'interno dei limiti del BIOS, ma rimappando in realtà la geometria virtuale su quella reale. Pur funzionando, questo approccio rovinava però il lavoro del sistema operativo, che prestava attenzione nel collocare i dati in modo da minimizzare i tempi di ricerca.

Le unità IDE evolsero successivamente nelle unità **EIDE** (*Extended IDE*) che supportavano anche un secondo schema d'indirizzamento chiamato **LBA** (*Logical Block Addressing*, "indirizzamento logico dei blocchi"), che semplicemente numerava i settori a partire da 0 e fino a un massimo di $2^{28} - 1$. Questo schema, anche se richiedeva che il controllore convertisse gli indirizzi LBA negli indirizzi della testina, del settore e del cilindro, permetteva di superare il limite dei 504 MB. Sfortunatamente questa modalità d'indirizzamento creò un nuovo collo di bottiglia corrispondente a $2^{28} \times 2^9$ byte (128 GB). Quando nel 1994 fu adottato lo standard EIDE, nessuno poteva però immaginare dischi di quelle dimensioni. I comitati degli standard, allo stesso modo dei politici, hanno la tendenza a rinviare nel tempo i problemi, in modo che sia il comitato successivo a doverli risolvere.

Le unità e i controllori EIDE presentavano anche altri miglioramenti. I controllori EIDE, per esempio, potevano avere due canali, ciascuno dei quali poteva avere un disco primario e uno secondario. Questa organizzazione ammetteva un massimo di quattro unità per controllore. Vennero supportati anche i lettori di CD-ROM e DVD e la velocità di trasferimento dati passò da 4 MB/s a 16,67 MB/s.

Mentre la tecnologia dei dischi continuava a migliorare, evolveva anche lo standard EIDE; per qualche strana ragione il suo successore fu però chiamato **ATA-3** (*AT Attachment*, "tecnologia avanzata per la connessione"), con un riferimento al PC/AT IBM (dove AT si riferiva alla allora Tecnologia Avanzata di una CPU a 16 bit che andava alla velocità di 8 MHz). Nella versione successiva lo standard fu chiamato **ATAPI-4** (*ATA Packet Interface*, "interfaccia per pacchetti ATA") e la velocità fu aumentata fino a 33 MB/s. Nello standard ATAPI-5 la velocità raggiunse invece i 66 MB/s.

A questo punto il limite dei 128 GB imposto dagli indirizzi LBA a 24 bit cominciò a profilarsi sempre più minaccioso, e quindi in ATAPI-6 la dimensione di LBA fu portata a 48 bit. Il nuovo standard finirà nei guai quando i dischi raggiungeranno la dimensione di $2^{48} \times 2^9$ byte. Con un aumento annuale del 50% della capacità, il limite dovuto ai 48 bit dovrebbe sopravvivere fino al 2035 circa (per scoprire la soluzione a questo problema si consulti l'undicesima edizione di questo libro!). I più astuti stanno già scommettendo su un aumento della dimensione di LBA a 64 bit. Lo standard ATAPI-6 ha anche aumentato la velocità di trasferimento dati a 100 MB/s e ha affrontato per la prima volta il problema del rumore dei dischi.

Lo standard ATAPI-7 rappresenta una rottura radicale con il passato. Questo standard, invece di aumentare la dimensione del connettore del disco (per migliorare la velocità di trasferimento), usa quella che viene chiamata interfaccia **serial ATA** per trasferire 1 bit alla volta su un connettore a 7 pin a una velocità che parte da 150 MB/s, ma che nel tempo dovrebbe raggiungere 1,5 GB/s. La sostituzione di un cavo piatto a 80 fili con un cavo rotondo di pochi millimetri di spessore migliora la circolazione dell'aria all'interno del calcolatore. Inoltre

l'interfaccia serial ATA utilizza 0,5 volt per i segnali (rispetto ai 5 V delle unità ATAPI-6), riducendo il consumo di energia. Il problema del consumo energetico dei dischi sta acquistando una grande importanza, sia nei sistemi di fascia alta, sia in quelli di fascia bassa, dove i portatili hanno un'alimentazione limitata (Gurumurthi et al., 2003).

2.3.5 Dischi SCSI

Per quanto riguarda l'organizzazione di cilindri, tracce e settori, i dischi SCSI (si pronuncia *scasi*) non sono differenti da quelli IDE, ma hanno una diversa interfaccia e una velocità di trasferimento dati molto più elevata. Le origini dei dischi SCSI risalgono al 1979 quando la società di Howard Shugart, l'inventore del floppy disk, introdusse il disco SASI (*Shugart Associates System Interface*, "sistema d'interfaccia Sugart Associates"). Nel 1986, dopo alcune modifiche e molte polemiche, venne standardizzato dalla ANSI e il suo nome cambiò in **SCSI** (*Small Computer System Interface*, "sistema d'interfaccia per piccoli sistemi"). A partire da quel momento sono state standardizzate versioni sempre più veloci con il nome di Fast SCSI (10 MHz), Ultra SCSI (20 MHz), Ultra2 SCSI (40 MHz), Ultra3 SCSI (80 MHz) e Ultra4 SCSI (160 MHz). Ciascuna di queste interfacce aveva anche una versione *wide* ("larga") a 16 bit; la Figura 2.22 mostra le combinazioni principali.

Per via della velocità di trasferimento più elevata, i dischi SCSI sono lo standard di molte workstation UNIX di Sun, HP, SGI e altri produttori. Sono anche i dischi standard dei Macintosh e dei PC Intel di fascia alta, soprattutto nel caso dei server di rete.

SCSI non è soltanto un'interfaccia per hard disk, ma è anche un bus al quale possono essere collegati un controllore e sette dispositivi. Questi possono comprendere uno o più hard disk SCSI, CD-ROM, masterizzatori, scanner, unità a nastro e altre periferiche SCSI. Ciascun dispositivo SCSI possiede un ID, compreso tra 0 e 7 (15 nel caso di *wide* SCSI), e due connettori, uno per l'input e uno per l'output. I cavi connettono in serie l'output di un dispositivo con l'input del successivo, come un filo di lucine natalizie. L'ultimo dispositivo della fila deve avere una terminazione per evitare che le riflessioni

Nome	Bit di dati	MHz del bus	MB/s
SCSI-1	8	5	5
Fast SCSI	8	10	10
Wide Fast SCSI	16	10	20
Ultra SCSI	8	20	20
Wide Ultra SCSI	16	20	40
Ultra2 SCSI	8	40	40
Wide Ultra2 SCSI	16	40	80
Ultra3 SCSI	8	80	80
Wide Ultra3 SCSI	16	80	160
Ultra4 SCSI	8	160	160
Wide Ultra4 SCSI	16	160	320

Figura 2.22 Parametri dei modelli SCSI.

nell'estremità del bus SCSI interferiscano con altri dati lungo il bus. In genere il controllore si trova su una scheda aggiuntiva all'inizio della catena, anche se ciò non è esplicitamente richiesto dallo standard.

Il più comune cavo per lo standard SCSI a 8 bit ha 50 contatti, 25 dei quali sono cavi di terra accoppiati uno a uno con altri 25 in modo da fornire un'eccellente immunità al rumore, necessaria per poter operare ad alte velocità. Dei 25 cavi 8 sono per i dati, 1 per la parità, 9 per il controllo, mentre i rimanenti sono per l'alimentazione oppure riservati per usi futuri. I dispositivi a 16 bit (e quello a 32 bit) richiedono un secondo cavo per trasportare i segnali aggiuntivi. I cavi possono avere una lunghezza di vari metri, permettendo l'utilizzo di hard disk, scanner e altre periferiche esterne.

I controllori SCSI possono funzionare come iniziatori o come destinatari di comunicazione. Di solito il controllore funziona come iniziatore e lancia comandi ai dischi e ad altre periferiche; questi comandi sono blocchi della dimensione massima di 16 byte che indicano al destinatario che cosa deve fare. I comandi e le risposte avvengono in fase, utilizzando vari segnali di controllo per delimitare la fase e arbitrare l'accesso al bus quando più dispositivi tentano di usarlo nello stesso momento. Questo arbitraggio è importante in quanto SCSI consente a tutti i dispositivi di funzionare contemporaneamente, aumentando potenzialmente le prestazioni in un ambiente in cui vi possono essere più processi attivi allo stesso tempo (per esempio UNIX o Windows XP); IDE ed EIDE ammettono invece un solo dispositivo attivo ad ogni istante.

2.3.6 RAID

Nel corso dell'ultimo decennio le prestazioni delle CPU sono aumentate in modo esponenziale, raddoppiando all'incirca ogni 18 mesi, mentre la stessa cosa non si è verificata per i dischi. Negli anni '70 il tempo medio di ricerca nei dischi dei minicomputer era compreso tra 50 e 100 ms, mentre ora è di 10 ms. In molti settori dell'industria (come quella automobilistica o aeronautica) un miglioramento delle prestazioni di un fattore 5 o 10 nell'arco di due decenni sarebbe una notizia di rilievo, ma nel settore dei computer è un dato imbarazzante. Per questo motivo la differenza tra le prestazioni della CPU e quella dei dischi è aumentata con il tempo in modo considerevole.

Come già visto, spesso si utilizza l'elaborazione parallela per velocizzare le prestazioni delle CPU. Quindi anche l'I/O parallelo è sembrato ad alcuni una buona idea. In un articolo del 1988 (Patterson et al., 1988) fu suggerita un'organizzazione a sei dischi per migliorare le prestazioni, l'affidabilità oppure entrambe allo stesso tempo. Questa idea fu adottata velocemente dal mercato e portò a una nuova classe di dispositvi di I/O chiamata **RAID**. Patterson e i suoi colleghi definirono **RAID** come **Redundant Array of Inexpensive Disks** ("insieme ridondante di dischi economici"), ma il mercato ridefinì la "I" di *Inexpensive* (economico) con il termine *Independent* (indipendente): forse per poter utilizzare dischi costosi? Dato che c'è sempre bisogno di un nemico (come in CISC contro RISC, anch'esso dovuto a Patterson), il ruolo dei cattivi fu assunto da **SLED** (*Single Large Expensive Disk*, "singolo disco capiente e costoso").

L'idea su cui si basa RAID è quella di installare vicino al calcolatore, di solito un server di grandi dimensioni, un contenitore pieno di dischi, di sostituire la scheda contenente il controllore del disco con un controllore RAID, di copiare i dati sul RAID e di continuare quindi le normali operazioni. In altre parole un RAID dovrebbe apparire agli occhi del sistema operativo come uno SLED, seppur con prestazioni e affidabilità più elevate. Dato che i dischi SCSI offrono buone prestazioni, prezzo basso e possibilità di avere fino a 7 unità su un singolo controllore (15 per l'interfaccia *wide* SCSI) è naturale che la maggior parte dei sistemi RAID consista di un controllore RAID SCSI, più un insieme di dischi SCSI che appaiono al sistema operativo come un'unica e grande unità. In questo modo non sono necessarie modifiche al software per poter usare RAID, il che rappresenta un grande vantaggio per gli amministratori di sistema.

Tutti i RAID, oltre ad apparire al software come un singolo disco, hanno la proprietà di distribuire i dati sulle diverse unità per permetterne la gestione parallela. Il gruppo di Patterson ha definito diversi schemi RAID, conosciuti con nomi che vanno da RAID livello 0 a RAID livello 5; esistono anche altri livelli di minor importanza che però non tratteremo. Il termine "livello" è in un certo modo fuorviante, in quanto non esiste alcuna gerarchia; si tratta semplicemente di sei diverse possibili organizzazioni.

Il RAID livello 0 è mostrato nella Figura 2.23(a). Secondo questa organizzazione il disco virtuale simulato dal RAID è visto come se ognuno dei k settori fosse diviso in *strip* ("strisce"), con i settori da 0 a $k - 1$ che compongono la *strip* 0, i settori da k a $2k - 1$ la *strip* 1 e così via; se $k = 1$ ogni *strip* è un settore, se $k = 2$ una *strip* è composta da due settori, ecc. L'organizzazione RAID livello 0 scrive sulle *strip* consecutive in modo ciclico (modalità *round robin*), come mostra la Figura 2.23(a) nel caso di un RAID con quattro unità. Questo modo di distribuire dati su più unità è chiamato **striping**. Se per esempio il software lancia un comando per leggere un blocco di dati costituito da quattro *strip* consecutive a partire dall'inizio di una *strip*, il controllore RAID spezzerà questo comando in quattro comandi separati, uno per ciascuno dei quattro dischi, e li farà eseguire in parallelo. In questo modo si ottiene un I/O parallelo senza che il software ne sia a conoscenza.

Il RAID livello 0 lavora meglio quando le richieste sono di grandi dimensioni. Se la dimensione di una richiesta supera il numero di unità moltiplicato per la dimensione di una *strip*, allora alcune unità riceveranno più richieste, che verranno soddisfatte una dopo l'altra. Spetta al controllore separare la richiesta e assegnare alle unità appropriate i comandi corretti nella giusta sequenza, oltre ad assemblare poi, correttamente, i risultati in memoria. Le prestazioni sono eccellenti e l'implementazione è semplice.

Il RAID livello 0 lavora peggio con i sistemi operativi che tendono a richiedere i dati un settore alla volta. In una simile situazione il risultato sarà comunque corretto, ma non vi sarà parallelismo da sfruttare e quindi non si otterrà alcun guadagno nelle prestazioni. Un altro svantaggio di questa organizzazione è che l'affidabilità è potenzialmente peggiore di quella che si ha con uno SLED. Se un RAID consiste di quattro dischi, ciascuno con un intervallo medio tra guasti di 20.000 ore, all'incirca ogni 5000 ore un disco si guasterà e i dati saranno interamente persi; uno SLED con un intervallo medio tra guasti di 20.000 ore sarebbe quattro volte più affidabile. Inoltre questo tipo di progettazione non è in realtà un vero e proprio RAID, in quanto non c'è alcuna ridondanza.

Il RAID livello 1, mostrato nella Figura 2.23(b), è invece un vero RAID in quanto duplica tutti i dischi, risultando così composto da quattro dischi primari e da quattro di backup. Nel caso di una scrittura ogni *strip* viene scritta due volte, mentre nel caso di una lettura è possibile usare entrambe le copie, distribuendo il carico di lavoro su più unità. Ne consegue che le prestazioni in scrittura non sono migliori di quelle che si otterrebbero con una singola unità, mentre quelle in lettura possono essere fino a due volte migliori. La tolleranza ai

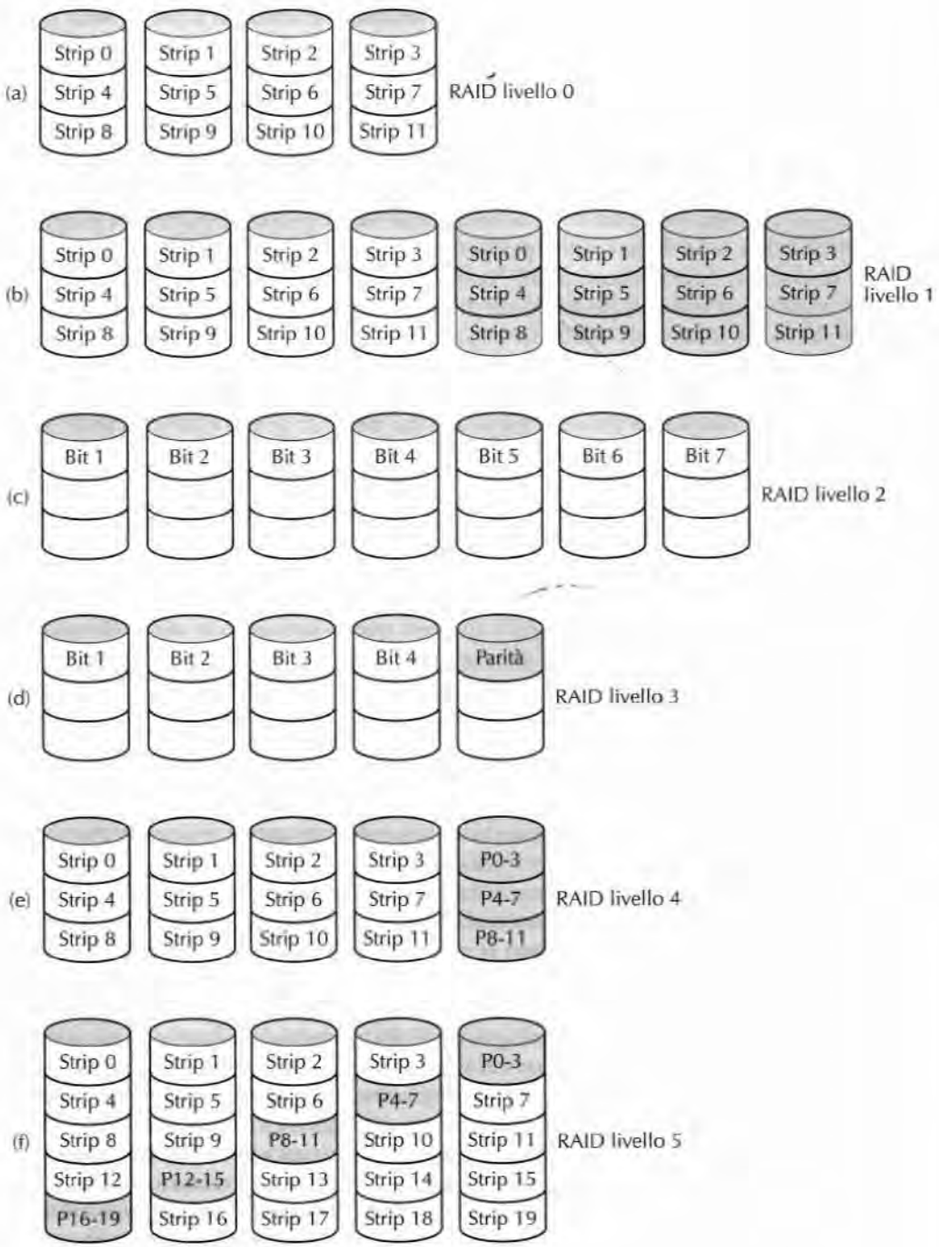


Figura 2.23 RAID dal livello 0 al livello 5. I dischi di backup e di parità sono in grigio.

guasti è eccellente: se un disco si rompe, si può semplicemente utilizzare la copia. La riparazione consiste nell'installare un nuovo disco e copiarvi l'intero disco di backup.

Diversamente dai livelli 0 e 1 che lavorano con *strip* di settori, il RAID livello 2 funziona sulla base di una parola o, in alcuni casi, anche sulla base di un byte. Immaginiamo di dividere ciascun byte del singolo disco virtuale in una coppia di *nibble* (mezzo byte) di 4 bit, e di aggiungere a ciascuno di loro un codice di Hamming per formare una parola di 7 bit, in cui i bit 1, 2 e 4 sono bit di parità. Immaginiamo inoltre che i sette dischi della Figura 2.23(c) siano sincronizzati per quanto riguarda le posizioni dei bracci e delle rotazioni. In questo modo sarebbe possibile scrivere una parola del codice di Hamming a 7 bit sui sette dischi, usando un bit per ogni disco.

Il calcolatore Thinking Machines CM-2 utilizzava questo schema, prendendo parole di dati a 32 bit, aggiungendovi 6 bit di parità per formare una parola di Hamming a 38 bit, oltre a un bit extra per la parità dell'intera parola, e distribuendo ciascuna parola su 39 dischi. Il *throughput* totale era estremamente elevato, dato che nel tempo necessario per accedere a un settore era possibile scrivere una quantità di dati pari a 32 settori. Inoltre la perdita di un disco non causava problemi, in quanto comportava la perdita di 1 solo bit per ciascuna parola a 39 bit che veniva letta, perdita che il codice di Hamming riesce a gestire autonomamente.

Questa organizzazione presenta però alcuni difetti: la rotazione dei dischi deve essere sincronizzata e lo schema ha senso soltanto se si utilizza un numero significativo di unità (anche con 32 dischi di dati e 6 dischi di parità l'overhead è del 19%). Inoltre si richiede molto lavoro al controllore, dato che deve calcolare la *checksum* del codice di Hamming a ogni tempo di bit.

Il RAID livello 3, mostrato nella Figura 2.23(d), è una versione semplificata del RAID livello 2. Il bit di parità viene calcolato per ogni parola di dati e poi scritto su un apposito disco. Dato che le parole di dati sono distribuite su più unità, anche in questo caso, come per il RAID livello 2, i dischi devono essere sincronizzati.

A prima vista potrebbe sembrare che un solo bit di parità possa permettere la sola rilevazione degli errori, ma non la loro correzione. Nel caso di errori casuali, questa osservazione è vera; tuttavia, nel caso di una rottura di un disco, lo schema permette la correzione completa degli errori singoli, in quanto la posizione del bit errato è conosciuta. Se un disco si guasta il controllore assume semplicemente che tutti i suoi bit valgano 0; se una parola ha un valore errato di parità, allora il bit del disco guastatosi doveva per forza valere 1 e poteva quindi essere corretto. Anche se i RAID livello 2 e livello 3 permettono un tasso di trasferimento dati elevato, il numero di richieste di I/O al secondo è lo stesso che per un singolo disco.

Anche i RAID livello 4 e livello 5 lavorano in base alle *strip*, e non a singole parole con parità; inoltre non richiedono la sincronizzazione tra i dischi. Il RAID livello 4 [vedi Figura 2.23(e)] è come il RAID livello 0, con una parità *strip*-per-*strip* scritta su un disco aggiuntivo. Per esempio, se ogni *strip* è lunga k byte, si calcola l'OR ESCLUSIVO fra tutte, ottenendo una *strip* di parità lunga k byte. Se un disco si guasta è possibile ricalcolare i byte persi grazie al disco di parità.

Questo schema protegge dalla perdita di un disco, ma ha prestazioni scarse quando si aggiornano piccole quantità di dati. Infatti, se si modifica un settore è necessario leggere tutti i dischi, in quanto occorre ricalcolare e riscrivere la parità. Una soluzione alternativa

consiste nel leggere i precedenti dati e valori di parità e, a partire da questi, calcolare quelli nuovi. Anche ricorrendo a questa ottimizzazione una piccola modifica dei dati richiede comunque due letture e due scritture; un'organizzazione dei dischi di questo tipo è chiaramente inefficiente.

Il disco di parità può inoltre diventare un collo di bottiglia, a causa del grande carico di lavoro che pesa su di esso. Il RAID livello 5 elimina questo collo di bottiglia distribuendo uniformemente i bit di parità su tutti i dischi, in modalità *round robin*, com'è mostrato nella Figura 2.23(f). Tuttavia, quando si verifica un guasto a un disco, la ricostruzione del suo contenuto è un processo complesso.

2.3.7 CD-ROM

I dischi ottici furono originariamente sviluppati per registrare programmi televisivi, ma al giorno d'oggi si prestano a utilizzi più interessanti se vengono usati come dispositivi di memorizzazione per calcolatori. I dischi ottici, grazie alla loro grande capacità e al basso costo, sono ampiamente utilizzati per distribuire software, libri, film e dati di tutti i tipi, nonché per creare copie di backup degli hard disk.

La prima generazione di dischi ottici fu inventata dalla multinazionale olandese Philips per memorizzare film. Avevano un diametro di 30 cm ed erano prodotti sotto il nome di LaserVision; non presero mai piede, se non in Giappone.

Nel 1980 la Philips sviluppò insieme alla Sony il CD (Compact Disc), che sostituì rapidamente il disco in vinile 33 giri per la registrazione della musica. I dettagli tecnici del CD vennero pubblicati in uno Standard Internazionale ufficiale (IS 10149), chiamato confidenzialmente **Libro rosso** (*Red Book*) per via del colore della copertina. (Gli Standard Internazionali sono pubblicati dalla *International Organization for Standardization*, la controparte internazionale dei gruppi nazionali per gli standard, come ANSI, DIN, e così via. Ognuno ha un proprio numero IS.) La scelta di pubblicare le specifiche dei dischi e dei lettori come uno Standard Internazionale fu presa per permettere di funzionare insieme ai CD venduti da diverse società discografiche e ai lettori di diversi produttori di elettronica. Tutti i CD hanno un diametro di 120 mm e uno spessore di 1,2 mm, con un buco di 15 mm nel centro. Il CD audio è stato il primo supporto digitale di memorizzazione di massa. I CD hanno una durata di vita prevista di circa 100 anni; ricordatevi quindi di eseguire, nel 2080, un controllo per verificare la qualità della prima partita di dischi!

La preparazione di un CD avviene per mezzo di un laser infrarosso ad alta potenza che crea sulla superficie fotosensibile di un disco di vetro (*glass master*) dei buchi dal diametro di 0,8 micron. A partire da questo *master* viene creato uno stampo che presenta rilievi in corrispondenza delle scanalature prodotte dal laser e all'interno del quale si inserisce del policarbonato liquido. Il risultato è la creazione di un CD con le scanalature disposte esattamente come quelle del *glass master*. Successivamente si deposita sul policarbonato un sottile strato di alluminio riflettente, lo si ricopre con una vernice protettiva e infine con un'etichetta. Le scanalature nel sottostrato di policarbonato sono chiamate **pit**, mentre le aree non incise tra i *pit* sono chiamate **land**.

Per la lettura, un diodo laser a bassa potenza invia sui *pit* e sui *land* luce infrarossa con una lunghezza d'onda di 0,78 micron. Dato che il laser si trova sul lato del policarbonato i *pit* sporgono nella sua direzione come rilievi sulla superficie altrimenti piatta. Dato che i *pit* hanno un'altezza pari a un quarto della lunghezza d'onda della luce del laser, la luce riflessa

da un *pit* viene sfasata di mezza lunghezza d'onda rispetto alla luce riflessa sulla superficie circostante. Il risultato è che le due parti interferiscono in modo distruttivo e restituiscono al fotorilevatore del lettore meno luce di quanta viene riflessa in corrispondenza dei *land*. In questo modo il lettore può distinguere un *pit* da un *land*. Anche se l'approccio più semplice potrebbe sembrare quello di usare un *pit* per registrare uno 0 e un *land* per memorizzare un 1, è più affidabile codificare i valori 1 e 0 come la presenza o l'assenza di una transizione *pit/land* o *land/pit*.

I *pit* e i *land* sono scritti in un'unica spirale continua (Figura 2.24) che parte vicino al buco e giunge a una distanza di 32 mm dal bordo esterno dopo aver girato 22.118 volte attorno al disco (circa 600 volte per millimetro). Se tale spirale venisse srotolata coprirebbe una lunghezza di 5,6 Km.

Per poter riprodurre la musica a una velocità uniforme è necessario che i *pit* e i *land* passino sotto la testina a una velocità lineare costante. Di conseguenza la velocità angolare del CD deve essere ridotta in modo continuo quando la testina di lettura si sposta dal centro del CD verso l'esterno. All'interno la velocità angolare è di 530 giri/min, corrispondente alla velocità lineare desiderata di 120 cm/s; all'esterno, per ottenere la stessa velocità lineare, la rotazione scende a 200 giri/min. C'è quindi molta differenza tra un'unità a velocità lineare costante e un disco magnetico, che funziona invece a velocità angolare costante. Inoltre la velocità di 530 giri/min è decisamente lontana da quella della maggior parte dei dischi magnetici, compresa tra i 3600 e i 7200 giri/min.

Nel 1984 Philips e Sony compresero il potenziale dell'uso dei CD per memorizzare i dati dei calcolatori e pubblicarono così il **Libro giallo** (*Yellow Book*) che definiva un preciso standard per quelli che ora vengono chiamati **CD-ROM** (*Compact Disc-Read Only Memory*). Per continuare a supportare il mercato dei CD audio, già di grandi dimensioni, si scelse di definire i CD-ROM con le stesse dimensioni fisiche dei CD audio, di renderli compatibili meccanicamente e otticamente e di produrli utilizzando le stesse macchine da stampo con iniezione di policarbonato. Questa decisione obbligò a utilizzare lenti motori a velocità va-

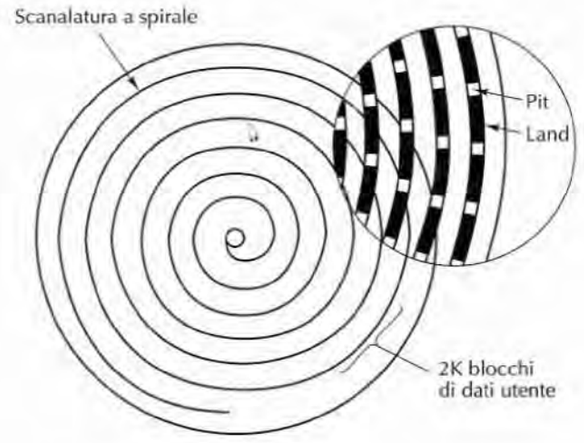


Figura 2.24 Struttura di registrazione di Compact Disc e CD-ROM.

riabile, ma permise allo stesso tempo di mantenere il costo di produzione di un CD-ROM ben al di sotto di un dollaro per volumi di medie dimensioni.

Il Libro giallo si occupò di definire una formattazione per i dati e migliorò le capacità di correzione degli errori del sistema. Ciò fu un passo fondamentale dato che, anche se gli amanti della musica non si preoccupano di perdere un bit ogni tanto, gli amanti dei computer tendono invece a essere molto schizzinosi al riguardo. Il formato base di un CD-ROM consiste nel codificare ogni byte in un simbolo a 14 bit. Come abbiamo già visto, 14 bit sono sufficienti per applicare un codice di Hamming a un byte, avendo a disposizione ancora 2 bit liberi; in realtà viene utilizzato un sistema di codifica più potente. In fase di lettura la corrispondenza 14/8 è realizzata in hardware mediante una tabella di ricerca.

Salendo di un livello, un gruppo di 42 simboli consecutivi forma un **frame** di 588 bit; ciascuno dei quali contiene 192 bit (24 byte) di dati. I restanti 396 bit sono usati per la correzione degli errori e per il controllo. Fino a questo punto lo schema è identico sia per i CD audio sia per i CD-ROM.

Il Libro giallo aggiunge un raggruppamento di 98 frame in quello che viene chiamato settore del CD-ROM, com'è mostrato nella Figura 2.25. Ogni settore del CD-ROM inizia con un preambolo di 16 byte, di cui i primi 12 contengono la sequenza (esadecimale) 00FFFFFFFFFFFFFFFFFFFF00, che permette al lettore di riconoscere l'inizio di un settore del CD-ROM. I 3 byte seguenti contengono il numero del settore, necessario dato che la ricerca all'interno dell'unica spirale di dati di un CD-ROM è molto più difficile di quella lungo le tracce concentriche di un disco magnetico. Per effettuare la ricerca il software del lettore calcola in modo approssimato dove andare, sposta la testina in quel punto e comincia quindi a cercare un preambolo nelle vicinanze, per capire quanto buona fosse la sua stima iniziale. L'ultimo byte del preambolo contiene il *modo*.

Il Libro giallo definisce due modi. Il Modo 1 usa la struttura della Figura 2.25, con un preambolo di 16 byte, 2048 byte di dati e un codice di errore di 288 byte (un codice Reed-Solomon incrociato e intervallato). Il Modo 2 combina i dati e i campi ECC in un campo dati di 2336 byte per quelle applicazioni che non richiedono (o non possono permettersi di eseguire) la correzione degli errori, come l'audio e il video. Occorre notare che per fornire un'affidabilità eccellente vengono usati tre diversi schemi per la correzione degli errori: all'interno di un simbolo, all'interno di un frame e all'interno di un settore del CD-ROM. Gli errori singoli sono corretti al livello più basso, gli errori relativi a piccoli flussi di dati sono corretti al livello del frame e quelli rimanenti sono gestiti dal livello del settore. Il prezzo pagato per questa affidabilità è la richiesta di 98 frame di 588 bit (7203 byte) per un campo dati di 1048 byte, con un'efficienza del 28%.

I lettori di CD-ROM a singola velocità funzionano a 75 settori/s, che corrisponde a una velocità di trasferimento dati di 153.600 byte/s nel Modo 1 e 175.200 byte/s nel Modo 2. I lettori a doppia velocità sono due volte più veloci, e lo stesso ragionamento vale per quelli a velocità più elevate. Un CD audio standard ha spazio per 74 minuti di musica, che, se usato per i dati in Modo 1, fornisce una capacità di 681.984.000 byte. Questa quantità è generalmente riportata come 650 MB, dato che 1 MB è composto da 2^{20} byte (1.048.576 byte) e non da 1.000.000 byte.

Occorre notare che neanche un lettore CD-ROM 32x (4.915.200 byte/s) raggiunge la velocità di un lettore SCSI-2 di dischi magnetici a 10 MB/s, anche se molti lettori CD-ROM usano l'interfaccia SCSI (esistono anche lettori CD-ROM IDE). Se si pensa che il tempo

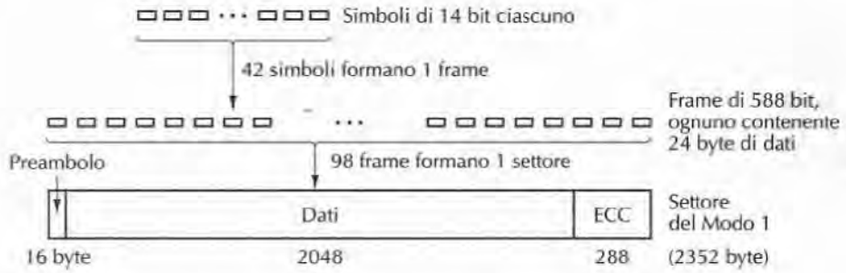


Figura 2.25 Struttura logica dei dati di un CD-ROM.

di ricerca è spesso di alcune centinaia di millisecondi, appare chiaro che i lettori CD-ROM, nonostante la loro grande capacità, non appartengono alla stessa classe di prestazione dei lettori di dischi magnetici.

Nel 1986 Philips stupì nuovamente con il **Libro verde** (*Green Book*), aggiungendo la grafica e una caratteristica fondamentale per i CD-ROM multimediali, ovvero la possibilità di combinare audio, video e dati in uno stesso settore.

L'ultima tessera del puzzle del CD-ROM è il suo file system. Per poter utilizzare lo stesso CD-ROM su calcolatori di tipo differente, fu necessario trovare un accordo sul file system da utilizzare. Per trovare questo accordo i rappresentanti di molte società di computer si incontrarono a Lake Tahoe sulle High Sierras, al confine tra California e Nevada e inventarono un file system che chiamarono **High Sierra**, divenuto in seguito uno Standard Internazionale (IS 9660). È composto da 3 livelli. Il livello 1 usa i nomi dei file fino a un massimo di 8 caratteri, eventualmente seguiti da un'estensione di 3 caratteri (la convenzione di MS-DOS per i nomi dei file). I nomi dei file possono contenere solo lettere maiuscole, cifre e il carattere trattino basso (*underscore*). Le directory possono essere annidate fino a una profondità massima di otto livelli, ma i loro nomi non possono contenere estensioni. Il livello 1 richiede che tutti i file siano contigui, ma ciò non rappresenta un problema per un supporto che viene scritto una sola volta. Ogni CD-ROM conforme al livello 1 IS 9660 può essere letto utilizzando MS-DOS, un calçolatore Apple, un computer UNIX o un qualsiasi altro calcolatore; per i produttori di CD-ROM questa proprietà rappresenta un vantaggio notevole. Il livello 2 ISO 9660 consente nomi lunghi fino a 32 caratteri, mentre il livello 3 permette file non contigui. Le estensioni Rock Bridge (il cui nome deriva, in modo stravagante, da quello della città del film *Mezzogiorno e mezzo di fuoco* di Mel Brooks) permette nomi molto lunghi (per UNIX), UID, GID e link simbolici; i CD-ROM non conformi al livello 1 non sono però leggibili da tutti i calcolatori.

2.3.8 CD-registrabili

Inizialmente l'attrezzatura necessaria per produrre il master di un CD-ROM (o, analogamente, un CD audio) era estremamente costosa, ma com'è solito avvenire nel mondo dei computer, nulla rimane costoso per lungo tempo. A partire da metà degli anni '90 i masterizzatori CD, grandi quanto i lettori CD, divennero una periferica comune venduta

nei negozi di computer. Questi dispositivi erano diversi dai dischi magnetici dato che i CD-ROM, una volta scritti, non potevano essere cancellati. Nonostante questo limite si ritagliarono uno spazio come supporto sul quale effettuare il backup di hard disk di grandi dimensioni; inoltre permisero a privati o a giovani società di produrre piccole tirature dei propri CD-ROM o di realizzare delle copie master da consegnare agli impianti professionali addetti alla duplicazione di CD in grandi volumi. Questi supporti sono conosciuti come **CD-R** (**CD-Registrabili**).

Dal punto di vista fisico i CD-R sono composti da un disco di policarbonato di 12 cm simile a quello dei CD-ROM, tranne per il fatto che contengono una scanalatura larga 0,6 mm che serve a guidare il laser nella fase di scrittura. La scanalatura ha un andamento sinusoidale di 0,3 mm con una frequenza esatta di 22,05 kHz per permettere un feedback continuo grazie al quale è possibile monitorare accuratamente la velocità di rotazione e, se necessario, modificarla. I primi CD-R assomigliavano ai normali CD-ROM tranne per il fatto che la parte alta era dorata invece che argentata; il colore dorato derivava dal fatto che, al posto dell'alluminio, veniva utilizzato vero oro per lo strato riflettente. Sui CD-R, diversamente dai CD argentati sui quali vi sono delle vere scanalature fisiche, le diverse proprietà riflettenti dei *pit* e dei *land* devono essere simulate. Ciò è realizzato aggiungendo uno strato di pigmento tra il policarbonato e il livello riflettente, come mostra la Figura 2.26. Vengono usati due tipi di pigmenti: la cianina, di colore verde, e la ftalocianina, di colore arancione giallastro; i chimici possono intavolare interminabili dibattiti su quale sia il migliore. Questi pigmenti sono simili a quelli usati in fotografia, e ciò spiega perché Kodak e Fuji siano fra i principali produttori di CD-R. In alcuni casi uno strato di alluminio riflettente sostituisce quello d'oro.

Lo strato di pigmento, nello stato iniziale, è trasparente e permette alla luce del laser di passarvi attraverso e di essere riflessa dallo strato riflettente. Per scrivere sul CD-R la potenza del laser viene portata a un valore alto, tra 8 e 16 mW. Quando il fascio colpisce una regione del pigmento, esso lo scalda al punto da rompere un legame chimico e questo cambiamento della struttura molecolare crea una regione scura. In fase di lettura (a 0,5 mW) il fotorilevatore vede una differenza tra le regioni scure in cui il pigmento è stato colpito e le aree trasparenti dove è ancora intatto. Quando il disco è letto da un normale lettore CD-ROM, o anche da un lettore di CD audio, questa differenza è interpretata allo stesso modo di quella che vi è tra i *pit* e i *land*.

Dato che ogni nuovo tipo di CD vuol vantare il proprio libro colorato, i CD-R ebbero il **Libro arancione** (*Orange Book*), pubblicato nel 1989. Questo documento definisce i CD-R e anche un nuovo formato, il **CD-ROM XA**, che permette di scrivere sui CD-R in modo incrementale: alcuni settori oggi, altri domani e altri ancora il mese successivo. Un insieme di settori scritti in una volta è chiamato **traccia del CD-ROM**.

Uno dei primi utilizzi del CD-R fu il PhotoCD di Kodak. In questo sistema il cliente portava un rullino fotografico e il proprio vecchio PhotoCD dal fotografo e ne usciva con lo stesso PhotoCD contenente, oltre alle vecchie foto, anche quelle nuove. Il nuovo gruppo di foto, creato scannerizzando i negativi, veniva scritto nel PhotoCD come una traccia separata del CD-ROM. La scrittura incrementale era una caratteristica necessaria, poiché i CD-R vergini erano troppo costosi per poterne utilizzare uno nuovo per ogni rullino fotografico.

La scrittura incrementale pone tuttavia nuovi problemi. Prima del Libro arancione tutti i CD-ROM avevano un unico **VTOC** (*Volume Table of Contents*, "indice del disco") al-

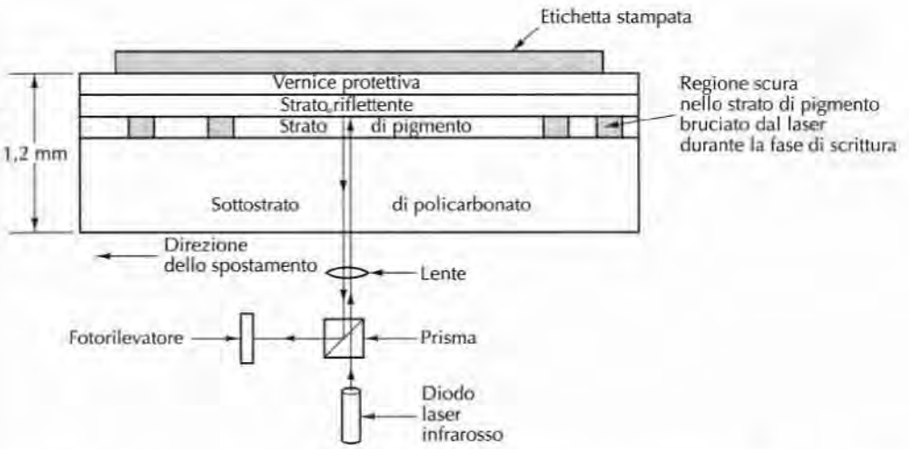


Figura 2.26 Spaccato di un disco CD-R e del laser (non in scala). La struttura di un CD-ROM è simile, fatta eccezione per la mancanza dello strato di pigmento e per la presenza di uno strato di alluminio con *pit* al posto dello strato riflettente.

l'inizio del disco; uno schema che non può però funzionare con le scritture incrementali (cioè multi-traccia). La soluzione adottata dal Libro arancione consiste nel dare a ogni traccia del CD-ROM il proprio VTOC, il cui elenco di file può comprendere alcuni o tutti i file presenti nelle tracce precedenti. Dopo che il CD-R è stato inserito nel lettore, il sistema operativo cerca fra tutte le tracce del CD-ROM per individuare il VTOC più recente, che descrive lo stato corrente del disco. Includendo nel VTOC corrente non tutti, ma solo alcuni dei file delle tracce precedenti, è possibile dare l'illusione che alcuni di loro siano stati cancellati. Le tracce possono essere raggruppate in **sessioni**; i CD-ROM **multisessione** non possono però essere letti da lettori standard di CD audio, dato che questi si aspettano la presenza di un'unica VTOC all'inizio del disco.

Ogni traccia deve essere scritta in un'unica operazione e in modo continuo senza interruzione; per questo motivo gli hard disk da cui sono prelevati i dati devono essere sufficientemente veloci per fornirli nei tempi corretti. Se i file da copiare sono sparsi nell'hard disk, i tempi di ricerca su disco possono far diminuire il flusso di dati al CD-R fino a provocare un temuto errore di esaurimento del buffer. Il risultato di un esaurimento del buffer è un piacevole e scintillante (anche se un po' costoso) sottobicchiere per i propri drink oppure un frisbee dorato o argentato di 120 mm! Di solito i programmi per i CD-R mettono a disposizione la possibilità di raggruppare tutti i file di input in una singola area contigua di 650 MB, rappresentante l'immagine del CD-ROM, prima di masterizzare il CD-R; questo processo in genere raddoppia i tempi effettivi di scrittura, richiede 650 MB di spazio libero su disco e, in ogni caso, non protegge da quegli hard disk che, non appena si riscaldano troppo, si fanno "prendere dal panico" e decidono di effettuare una ricalibrazione termica.

I CD-R permettono ai privati e alle società di copiare facilmente i CD-ROM (e i CD audio), spesso in violazione dei diritti d'autore. Sono stati ideati vari schemi per ostacolare

questa pirateria e far sì che sia difficile leggere un CD-ROM se non utilizzando il software fornito dall'autore dello stesso. Uno di questi sistemi prevede di registrare le lunghezze di tutti i file del CD-ROM come se fossero di vari gigabyte, ostacolando ogni tentativo da parte dei normali software di copiatura di trasferirli su hard disk. Le vere lunghezze sono memorizzate all'interno del software del produttore oppure sono nascoste (possibilmente criptate) in un'area inaspettata all'interno del CD-ROM. Un altro schema utilizza intenzionalmente ECC errati in specifici settori, con l'aspettativa che il programma di copiatura li "correggerà". Il software dell'applicazione controllerà anch'esso gli ECC e si rifiuterà di funzionare se essi sono corretti. Altre possibilità prevedono l'utilizzo di intervalli non standard tra le tracce e lo sfruttamento di altri "difetti" fisici.

2.3.9 CD-riscrivibili

Anche se abitualmente vengono usati supporti che permettono un'unica scrittura, come la carta o la pellicola fotografica, c'è una grande richiesta di CD-ROM riscrivibili. Una tecnologia oggi disponibile è quella dei **CD-RW** (*CD-ReWritable*, CD-Riscrivibili), che usano supporti della stessa dimensione dei CD-R. I CD-RW però, invece di usare la cianina o la ftalocianina per lo strato sul quale registrare, impiegano una lega di argento, indio, antimonio e tellurio. Questa lega ha due stati stabili, quello cristallino e quello amorfo, con diverse proprietà riflettenti.

I lettori CD-RW utilizzano laser che funzionano a tre potenze distinte. Alla potenza più elevata il laser scioglie la lega portandola dallo stato cristallino altamente riflettente a quello amorfo, dotato di una riflettività minore e che rappresenta un *pit*. Alla potenza media la lega si scioglie e si ricompone nello stato cristallino naturale per tornare a rappresentare un *land*. Alla potenza più bassa è possibile rilevare lo stato del materiale (per operazioni di lettura), senza però indurre alcuna trasformazione.

I CD-RW vergini sono molto più costosi dei CD-R vergini e per questo non li hanno sostituiti completamente. Inoltre il fatto che, una volta scritto, un CD-R non possa essere cancellato accidentalmente, rappresenta un gran vantaggio per il backup dei dischi.

2.3.10 DVD

Il formato base dei CD/CD-ROM è stato definito attorno al 1980. Da allora la tecnologia è migliorata, al punto che oggi è possibile realizzare dischi ottici di maggiore capacità in modo economicamente conveniente e c'è una grande richiesta per questo tipo di supporti. Hollywood amerebbe fervidamente sostituire le videocassette analogiche con i dischi digitali, dato che hanno una qualità più elevata, sono più economici da produrre, durano più a lungo, occupano meno spazio sugli scaffali dei negozi e non devono essere riavvolti. Le aziende di elettronica stanno cercando un nuovo prodotto per visualizzare i successi cinematografici e molte società informatiche vorrebbero aggiungere funzionalità multimediali al proprio software.

Questa combinazione tra tecnologia e domanda proveniente da tre mercati estremamente ricchi e potenti ha portato al **DVD**, che originalmente era l'acronimo di *Digital Video Disk* (disco video digitale), mentre ora significa ufficialmente *Digital Versatile Disk* (disco digitale versatile). I DVD sono progettati in modo simile ai CD: si inietta policarbonato in uno stampo e sulla superficie si possono distinguere *pit* e *land* che vengono illuminati da un diodo laser e letti da un fotorilevatore. Le novità riguardano l'uso di:

1. *pit* più piccoli (0,4 μ contro gli 0,8 dei CD)
2. una spirale più stretta (0,74 μ di distanza tra le tracce rispetto agli 1,6 dei CD)
3. un laser rosso (a 0,65 μ invece che a 0,78 come nei CD).

Insieme, questi miglioramenti permettono una capacità sette volte maggiore, che arriva fino a 4,7 GB. Un DVD 1x ha una velocità di 1,4 MB/s (contro i 150 KB/s dei CD). Sfortunatamente il passaggio ai laser rossi, come quelli usati nei supermercati, obbliga i lettori DVD ad avere un secondo laser o una sofisticata ottica di conversione per poter leggere i CD e i CD-ROM esistenti, cosa che non tutti i lettori permettono. Inoltre la lettura di CD-R e CD-RW può non essere possibile sui lettori DVD.

La capacità di 4,7 GB è sufficiente? Forse. Utilizzando la compressione MPEG-2 (standardizzata in IS 13346) un disco DVD da 4,7 GB può memorizzare 133 minuti di video a schermo intero in alta risoluzione (720 × 480), più di otto diverse colonne sonore e sottotitoli in 32 lingue diverse. Circa il 92% dei film che Hollywood ha finora prodotto ha una durata inferiore ai 133 minuti. Tuttavia, dato che alcune applicazioni, come i giochi multimediali o le enciclopedie, potrebbero richiedere quantità di dati più elevate e dato che Hollywood vorrebbe inserire più film sullo stesso disco, sono stati definiti quattro formati:

1. singolo lato, singolo strato (4,7 GB)
2. singolo lato, doppio strato (8,5 GB)
3. doppio lato, singolo strato (9,4 GB)
4. doppio lato, doppio strato (17 GB).

Perché così tanti formati? In una parola sola: politica. Philips e Sony volevano dischi a singolo lato e doppio strato per le versioni ad alta capacità, mentre Toshiba e Time Warner volevano dischi a doppio lato e doppio strato. Philips e Sony ritenevano che la gente non fosse disposta a girare i dischi per cambiar lato, mentre Time Warner non credeva che potesse funzionare l'utilizzo di due strati su un solo lato. Il compromesso fu quello di adottare tutte le soluzioni e di lasciare al mercato il compito di determinare quale sarebbe sopravvissuta.

La tecnologia a doppio strato ha uno strato riflettente nella parte bassa, coperto da uno semiriflettente. A seconda del punto in cui il laser è messo a fuoco, esso rimbalza su un livello oppure sull'altro. Il livello più basso richiede che i *pit* e i *land* siano leggermente più grandi per essere leggibili in modo affidabile, e per questo motivo la sua capacità è leggermente inferiore rispetto a quella dello strato superiore.

I dischi a doppio lato sono creati prendendo due dischi a singolo lato da 0,6 mm di spessore e incollandoli l'uno contro l'altro, cioè "schiena contro schiena". Per far sì che tutte le versioni abbiano lo stesso spessore, un disco a singolo lato è composto da un disco spesso 0,6 mm incollato con un sottostrato vuoto (o forse, in futuro, con uno contenente 133 minuti di pubblicità, nella speranza che la gente sia curiosa e giri il disco per vedere che cosa c'è sull'altro lato). La Figura 2.27 illustra la struttura del disco a doppio strato e doppio lato.

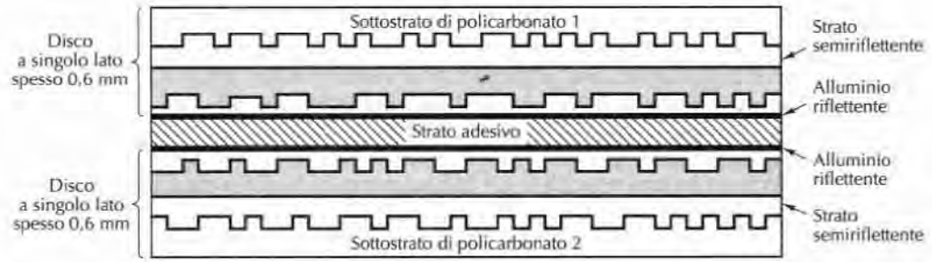


Figura 2.27 Disco DVD a doppio lato e doppio strato.

Il DVD è stato ideato da un consorzio di 10 aziende produttrici, sette delle quali giapponesi, in stretta collaborazione con i principali studi di produzione di Hollywood (alcuni dei quali sono di proprietà delle aziende giapponesi facenti parte del consorzio). Le società di computer e delle telecomunicazioni non furono invitate a questo "picnic" e l'attenzione finì per concentrarsi sull'utilizzo del DVD per il noleggio e la vendita di film. Tra le funzionalità standard sono presenti, per esempio, l'omissione in tempo reale delle scene scabrose (per permettere ai genitori di rendere adatto ai bambini un film vietato ai minorenni), l'audio a sei canali e il supporto per il *Pan-and-Scan*. Quest'ultima funzionalità permette di decidere dinamicamente come tagliare le bande laterali dei film (il cui rapporto larghezza-altezza è di 3:2) per essere visualizzate sugli attuali schermi televisivi (che hanno un rapporto di forma di 4:3).

Un altro aspetto che molto probabilmente non sarebbe stato tenuto in conto dalle società di computer è la scelta intenzionale di rendere incompatibili i dischi previsti per gli Stati Uniti rispetto a quelli previsti per l'Europa, e anche per gli altri continenti. Hollywood ha preteso questa "caratteristica" poiché i nuovi film escono sempre prima negli Stati Uniti e vengono distribuiti in Europa solo quando negli Stati Uniti sono ormai in vendita i video.

L'idea era quella di impedire che i negozi di video europei potessero comprare i video negli Stati Uniti prima dell'uscita in Europa, riducendo quindi gli introiti delle sale cinematografiche. Se Hollywood fosse stata il motore del mercato dei calcolatori avremmo avuto floppy disk da 3,5 pollici negli Stati Uniti e floppy disk da 9 cm in Europa.

2.3.11 Blu-Ray

Nel mercato dei calcolatori nulla rimane fermo, tanto meno la tecnologia per la memorizzazione dei dati. Il DVD è stato appena introdotto e già il suo successore minaccia di renderlo obsoleto; si tratta del **Blu-Ray**, chiamato in questo modo poiché utilizza un laser blu invece di quello rosso dei DVD. I laser blu hanno una lunghezza d'onda più piccola di quelli rossi, il che permette una messa a fuoco più accurata e l'uso di *pit* e *land* più piccoli. I dischi Blu-Ray a singolo lato contengono circa 25 GB di dati; quelli a doppio lato 50 GB. La velocità di trasferimento dati è di circa 4,5 MB/s, buona per essere un disco ottico, ma ancora molto lontana da quella dei dischi magnetici (per esempio ATAPI-6 a 100 MB/s e Ultra4 SCSI a 320 MB/s). Ci si aspetta che alla fine Blu-Ray sostituirà i CD-ROM e i DVD, ma questo passaggio potrebbe richiedere degli anni.

2.4 Input/Output

Com'è stato detto all'inizio di questo capitolo, un calcolatore è composto da tre componenti principali: la CPU, le memorie e i dispositivi di **I/O**. Finora abbiamo analizzato la CPU e le memorie; adesso passiamo allo studio dei dispositivi di I/O e alla loro connessione con il resto del sistema.

2.4.1 Bus

Da un punto di vista fisico la struttura della maggior parte dei calcolatori e delle workstation è simile a quella mostrata nella Figura 2.28. L'organizzazione più comune consiste in una scatola metallica contenente una grande scheda di circuiti stampati nella parte bassa, chiamata **scheda madre** (scheda genitore, sembra più *politically correct*). La scheda madre contiene il chip della CPU, alcuni slot nei quali è possibile inserire i moduli DIMM e vari altri chip di supporto. Contiene inoltre un bus stampato lungo la sua lunghezza e alcune prese nelle quali è possibile inserire i connettori delle schede di I/O (il bus PCI). I PC più vecchi hanno anche un secondo bus (il bus ISA) per connettere le schede di I/O costruite molto tempo prima; di solito nei calcolatori più moderni questo bus non è presente e il suo utilizzo sta rapidamente scomparendo.

La struttura logica di un semplice personal computer di fascia bassa è mostrata nella Figura 2.29. Questo sistema ha un unico bus utilizzato per connettere la CPU, la memoria e i dispositivi di I/O, mentre la maggior parte dei sistemi ha invece due o più bus. Ciascun dispositivo di I/O è composto da due parti: una contenente la maggior parte dell'elettronica, chiamata il **controllore**, e un'altra contenente il dispositivo stesso, per esempio un lettore di dischi. Di solito il controllore è collocato su una scheda inserita in uno slot libero, tranne per quei controllori che non sono opzionali (come la tastiera) e che spesso sono installati direttamente sulla scheda madre. Anche se lo schermo (il monitor) non è opzionale, il controllore video è talvolta situato su una scheda aggiuntiva per permettere all'utente di scegliere tra schede con o senza acceleratori hardware, memoria aggiuntiva e altre caratteristiche avanzate. Il controllore è connesso al suo dispositivo mediante un cavo che si collega al connettore nella parte posteriore della scatola metallica.

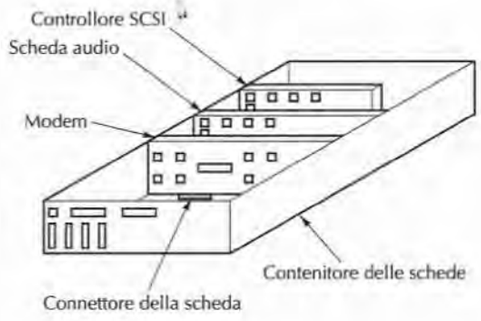


Figura 2.28 Struttura fisica di un personal computer.

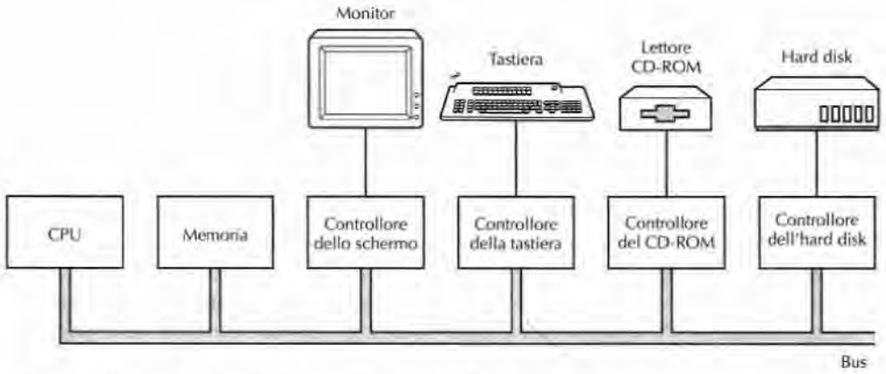


Figura 2.29 Struttura logica di un semplice personal computer.

Il compito di un controllore è di governare il proprio dispositivo di I/O e gestire il suo accesso al bus. Per esempio, quando un programma deve ottenere dei dati dal disco, manda un comando al suo controllore, che spedisce delle istruzioni al lettore, tra cui quella per la ricerca su disco. Il lettore, dopo aver individuato la traccia e il settore corretti, restituisce i dati al controllore sotto forma di un flusso seriale di bit. È compito del controllore spezzare il flusso di bit in unità, generalmente costituite da una o più parole, e scriverle successivamente in memoria. Quando un controllore legge e scrive dati in memoria senza l'intervento della CPU si dice che effettua un **Direct Memory Access** ("accesso diretto alla memoria"), meglio conosciuto con il suo acronimo **DMA**. Di solito, una volta che il trasferimento è completato, il controllore produce un **interrupt** che obbliga la CPU a sospendere immediatamente l'esecuzione del programma corrente e a far iniziare una speciale procedura, chiamata **gestore dell'interrupt** (*interrupt handler*) che controlla se ci sono errori, compie le azioni eventualmente necessarie e informa il sistema operativo che l'I/O è terminato. Dopo che il gestore dell'interrupt ha terminato il proprio lavoro, la CPU riprende l'esecuzione del programma precedentemente sospeso all'arrivo dell'interrupt.

Il bus non è utilizzato solamente dai controllori di I/O, ma anche dalla CPU quando deve prelevare istruzioni o dati. Che cosa succede se la CPU e un controllore di I/O cercano di usare il bus allo stesso tempo? Un chip chiamato **arbitro del bus** stabilisce i turni. Generalmente le periferiche di I/O hanno la precedenza sulla CPU, dato che i dischi e altri dispositivi mobili non possono essere fermati e obbligarli ad aspettare vorrebbe dire perdere dati. Quando nessuna operazione di I/O è in esecuzione, la CPU può utilizzare tutti i cicli del bus per accedere alla memoria. Tuttavia quando un dispositivo di I/O è già in funzione, se richiede l'accesso al bus, questo gli viene subito concesso. Questo processo si chiama **furto di cicli** e rallenta le prestazioni del calcolatore.

Questo schema funzionava bene per i primi personal computer, dato che tutti i componenti erano più o meno bilanciati. Tuttavia, con l'aumento della velocità di CPU, memorie e dispositivi di I/O, sorse un problema: il bus non era più in grado di gestire il carico affidatogli. In un sistema chiuso, come le workstation adibite a calcoli ingegneristici, la soluzione fu quella di progettare un nuovo bus più veloce per il modello successivo. Questo approccio funzionava correttamente, dato che nessuno pensava di riutilizzare i vecchi dispositivi di I/O con il nuovo sistema.

Nel mondo dei PC però gli utenti aggiornavano spesso le loro CPU, ma allo stesso tempo volevano continuare a utilizzare stampanti, scanner e modem anche con il nuovo sistema. Era nato inoltre un enorme mercato attorno alla fornitura di periferiche di ogni tipo per il bus del PC IBM e non si voleva certo buttar via tutti gli investimenti per ricominciare da capo. IBM interpretò questo fatto nel modo peggiore quando lanciò il successore del PC IBM, la linea PS/2. Il PS/2 era dotato di un nuovo bus più veloce, ma molti produttori di cloni continuarono a utilizzare il bus dei vecchi PC, ora chiamato bus **ISA** (*Industry Standard Architecture*, "standard industriale di architettura"). Molti produttori di dischi e di altre periferiche continuarono a costruire controllori per quel bus e così IBM si trovò nella paradossale situazione di essere l'unico produttore di PC non più compatibili con IBM. Alla fine fu obbligata a tornare a supportare il bus ISA. Aprendo una parentesi, occorre notare che ISA se utilizzato nel contesto dei livelli di una macchina significa *Instruction Set Architecture* ("architettura dell'insieme d'istruzioni"), mentre *Industry Standard Architecture* quando si parla di bus.

Ciononostante, anche se il mercato esercitava pressioni affinché nulla venisse modificato, i vecchi bus divennero realmente troppo lenti. Questa situazione portò altre società a sviluppare macchine dotate di più bus, tra le quali c'era ancora il vecchio bus ISA o il suo successore retrocompatibile **EISA** (*Extended ISA*, "ISA esteso"). Oggi il più diffuso di questi bus è il **PCI** (*Peripheral Component Interconnect*, "interconnessione di componenti periferici"), progettato da Intel. Al fine di incoraggiare l'intero mercato (compresi i propri concorrenti) ad adottarlo, Intel decise però di rendere di dominio pubblico tutti i brevetti.

Il bus PCI può essere utilizzato in varie configurazioni, di cui la più comune è illustrata nella Figura 2.30. In questo schema la CPU comunica con il controllore della memoria lungo una connessione dedicata ad alta velocità. Il controllore comunica con la memoria e con il bus PCI in modo diretto, per far sì che il traffico tra CPU e memoria non occupi il bus PCI. Le periferiche ad alta larghezza di banda (cioè alta velocità di trasferi-

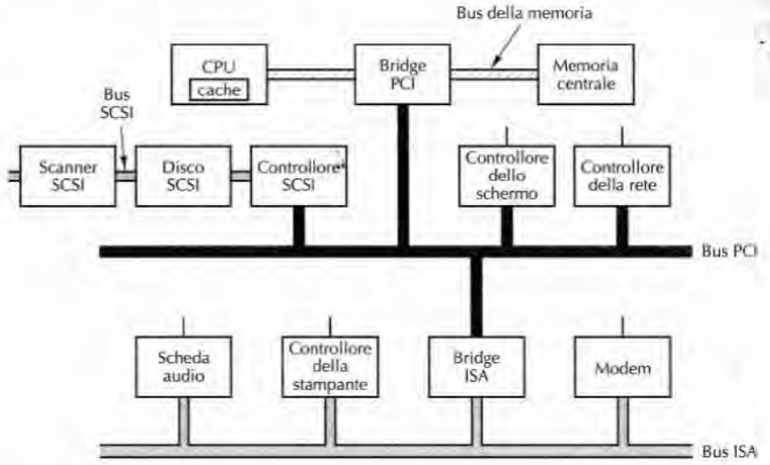


Figura 2.30 Tipico PC odierno con un bus PCI e un bus ISA. La scheda audio e il modem sono dispositivi ISA; il controllore SCSI è un dispositivo PCI.

mento dati) possono comunque connettersi direttamente al bus PCI. Inoltre, in questa configurazione, il bus PCI ha un ponte verso il bus ISA in modo da poter continuare a utilizzare i controllori ISA e i loro dispositivi, anche se, come detto precedentemente, questo bus sta gradualmente scomparendo. Di solito una macchina progettata in questo modo ha tre o quattro slot PCI liberi e uno o due slot ISA, per consentire agli utenti di collegare sia le vecchie schede ISA di I/O (generalmente per dispositivi lenti) sia le nuove schede PCI di I/O (di solito per periferiche più veloci).

Al giorno d'oggi sono disponibili molti tipi di dispositivi di I/O; ne tratteremo ora alcuni dei più comuni.

2.4.2 Terminali

I terminali sono composti da due parti principali: la tastiera e il monitor. Nel mondo dei mainframe spesso questi componenti sono integrati in un unico dispositivo, collegato al calcolatore principale mediante una linea seriale o un cavo telefonico. Ancor oggi questi dispositivi sono ampiamente usati per le prenotazioni delle compagnie aeree, nelle banche e, più in generale, nei sistemi basati su mainframe. Nel mondo dei personal computer la tastiera e il monitor sono invece dei dispositivi indipendenti; la tecnologia è in ogni caso la stessa.

Tastiere

Esistono vari tipi di tastiere. Il PC IBM originario era dotato di una tastiera che aveva, sotto ogni tasto, un pulsante a scatto che dava un feedback tattile e produceva un clic quando il tasto veniva premuto sufficientemente. Oggi i tasti delle tastiere più economiche hanno semplicemente dei contatti meccanici, mentre quelle di qualità più elevata hanno un foglio di materiale elastometrico (una specie di gomma) tra i tasti e il circuito stampato sottostante. In questi modelli vi è, sotto ogni tasto, una piccola calotta che si piega quando il tasto viene premuto a sufficienza; una minuscola regione di materiale conduttivo presente al suo interno permette di chiudere il circuito. In alcune tastiere, la pressione di un tasto fa scorrere un piccolo corpo metallico all'interno di una bobina conduttrice inducendo in essa una corrente rilevabile. Vengono utilizzati anche vari altri metodi, di tipo meccanico o elettromagnetico.

La pressione di un tasto di un PC genera un interrupt che stimola una parte del sistema operativo, chiamata *gestore dell'interrupt della tastiera*. Questa routine, leggendo un registro hardware della tastiera, ricava il numero (da 1 a 102) associato al tasto premuto. Anche quando si rilascia un tasto viene generato un interrupt. In questo modo se l'utente preme il tasto SHIFT, poi preme e rilascia il tasto M e infine rilascia il tasto SHIFT, il sistema operativo può capire che l'utente voleva digitare una "M" maiuscola invece che una "m" minuscola. La gestione di sequenze multi-tasto che coinvolgono i tasti SHIFT, CTRL e ALT è fatta interamente via software (compresa la famosa e detestata sequenza CTRL-ALT-CANC utilizzata per riavviare i PC IBM e i loro cloni).

Monitor CRT

Un monitor contiene un tubo a raggi catodici, **CRT** (*Cathode Ray Tube*), e il proprio alimentatore. Il CRT è dotato di un cannone elettronico che indirizza un fascio di elettroni contro uno schermo fosforescente posizionato vicino alla parte frontale del tubo, com'è mostrato nella Figura 2.31(a). I monitor a colori hanno un fascio di elettroni, per ciascun componente rosso, verde e blu. Durante una scansione orizzontale il fascio impiega circa 50 μs

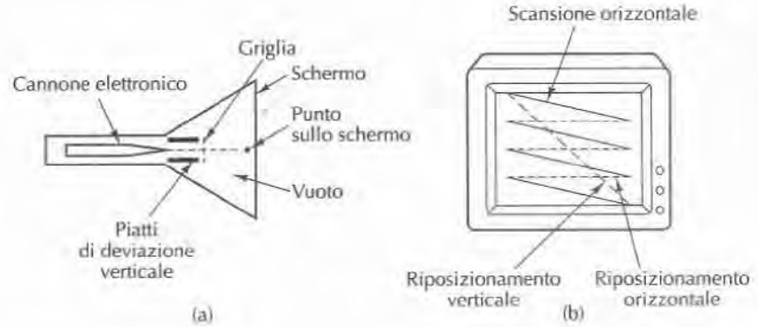


Figura 2.31 (a) Spaccato di un CRT. (b) Scansione di un CRT.

per scorrere lungo tutto lo schermo e disegnare una linea orizzontale. Successivamente esegue un nuovo tracciamento orizzontale in direzione opposta per riposizionarsi sul lato sinistro e poter iniziare una nuova passata. Un dispositivo di questo tipo che produce un'immagine linea per linea è chiamato dispositivo a **scansione raster**.

Per controllare la scansione orizzontale si applica una tensione linearmente crescente ai piatti collocati alla destra e alla sinistra del cannone elettronico, che servono a deviare orizzontalmente il fascio. Il moto verticale è controllato da un aumento molto più lento della tensione applicata ai piatti posizionati sopra e sotto il cannone. Dopo un numero di passate che varia all'incirca tra 400 e 1000, le tensioni sui piatti di deviazione orizzontali e verticali vengono invertite rapidamente per portare nuovamente il fascio nell'angolo in alto a sinistra dello schermo[6]. In genere un'immagine a pieno schermo è ridisegnata tra le 30 e le 60 volte al secondo, e in ognuna di queste il fascio di elettroni compie il percorso mostrato nella Figura 2.31(b). Anche se descrivendo il funzionamento dei CRT abbiamo detto che il fascio viene spostato sull'area dello schermo mediante campi elettrici, molti modelli, soprattutto nel caso di monitor professionali, usano al loro posto dei campi magnetici.

All'interno del CRT si trova una griglia, che permette di produrre sullo schermo un pattern costituito da punti. Quando si applica alla griglia una tensione positiva, gli elettroni vengono accelerati facendo sì che il fascio colpisca lo schermo emettendo luce per un breve istante. Quando invece si applica una tensione negativa gli elettroni vengono respinti, in modo che non passino attraverso la griglia e lo schermo non venga illuminato. La tensione applicata alla griglia permette quindi di far apparire sullo schermo il corrispondente pattern di bit; questo meccanismo consente di convertire un segnale elettrico binario in un'immagine visiva costituita da punti luminosi e punti scuri.

Schermi piatti

Dato che i CRT sono troppo ingombranti e pesanti per essere utilizzati nei calcolatori portatili, c'è bisogno di una tecnologia completamente diversa per i loro schermi. La tecnologia più comune è quella dello schermo a cristalli liquidi, **LCD** (*Liquid Crystal Display*).

---

6 Questo spostamento è spesso detto *fly back* (*N.d.Rev.*).

La sua descrizione verrà fortemente semplificata, dato che si tratta di una tecnologia particolarmente complessa, di cui esistono molte varianti e che cambia rapidamente.

I cristalli liquidi sono molecole organiche viscose che si muovono come un liquido, ma che hanno anche una struttura spaziale simile a quella di un cristallo. Scoperti nel 1888 da un botanico austriaco (Rheinitzer), negli anni '60 sono stati applicati per la prima volta agli schermi (per esempio in quelli di calcolatrici e orologi). Quando tutte le molecole sono allineate nella stessa direzione le proprietà ottiche del cristallo dipendono dalla direzione e dalla polarizzazione della luce incidente. È possibile modificare l'allineamento delle molecole, e quindi le proprietà ottiche, tramite l'applicazione di un campo elettrico. In particolare è possibile controllare elettricamente l'intensità della luce uscente facendola passare attraverso un cristallo liquido. Questa proprietà può essere sfruttata per costruire schermi piatti.

Uno schermo LCD consiste in due lastre di vetro parallele tra le quali è sigillato un cristallo liquido. Alle lastre sono attaccati degli elettrodi trasparenti. Una luce (naturale o artificiale) proveniente da dietro la lastra posteriore illumina lo schermo. Gli elettrodi trasparenti collegati a ciascuna lastra sono utilizzati per creare campi elettrici all'interno del cristallo liquido e, variando la tensione sulle diverse parti dello schermo, si può controllare l'immagine da rappresentare. Per la visualizzazione sono anche necessari dei filtri di polarizzazione incollati allo schermo frontale e a quello posteriore. Lo schema generale è mostrato nella Figura 2.32(a).

Anche se vengono usati diversi tipi di schermi LCD, ne considereremo come esempio solo uno: lo schermo **TN** (*Twisted Nematic*). In questo schermo la lastra posteriore contiene

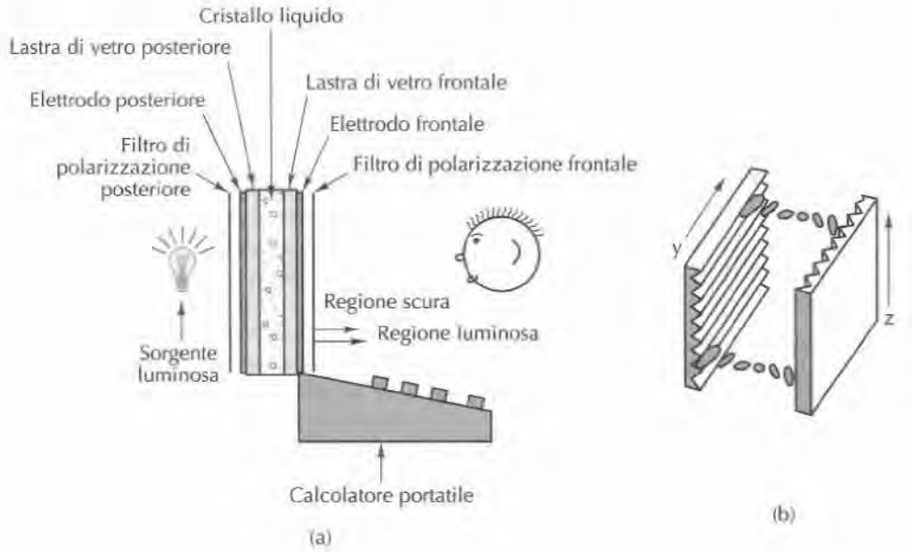


Figura 2.32 (a) Struttura di uno schermo LCD. (b) I solchi sulla lastra frontale e su quella posteriore sono perpendicolari tra loro.

piccoli solchi orizzontali, mentre in quella frontale i solchi sono disposti verticalmente, come mostra la Figura 2.32(b). In assenza di campo elettrico le molecole del LCD tendono ad allinearsi lungo i solchi e, dato che gli allineamenti sulle due lastre differiscono di 90 gradi, le molecole (e quindi la struttura del cristallo) effettuano una torsione nello spazio intermedio.

Dietro allo schermo è collocato un filtro di polarizzazione orizzontale, che permette esclusivamente il passaggio di luce polarizzata in quella direzione, mentre nella parte frontale c'è un filtro di polarizzazione verticale. Se tra le lastre non ci fosse del liquido, la luce lasciata passare dal filtro posteriore sarebbe bloccata da quello anteriore, rendendo lo schermo uniformemente nero.

La torsione della struttura del cristallo delle molecole guida la luce nel suo passaggio e ruota la sua polarizzazione rendendola verticale. Quindi, in assenza di un campo elettrico, lo schermo LCD risulta uniformemente luminoso. Applicando tensione a particolari regioni della lastra si distrugge la struttura cristallina e la sua torsione, bloccando localmente il passaggio della luce.

È possibile utilizzare due schemi per l'applicazione della tensione. Negli **schermi a matrice passiva** (economico) entrambi gli elettrodi contengono fili paralleli. Per esempio, in uno schermo 640 × 480 l'elettrodo posteriore potrebbe avere 640 fili verticali, e quello frontale 480 orizzontali. Applicando una tensione a uno dei cavi verticali e un impulso a uno dei cavi orizzontali si modifica il voltaggio di un pixel in una particolare posizione, facendola diventare scura per un breve intervallo di tempo. Applicando lo stesso impulso al pixel successivo e poi a quello ancora seguente si può disegnare sullo schermo una linea, in modo simile a quanto avviene in un CRT. Solitamente lo schermo viene interamente ridisegnato 60 volte in un secondo, in modo da ingannare l'occhio e fargli credere che stia guardando un'immagine costante; anche sotto questo aspetto il principio è lo stesso di un monitor CRT.

L'altro schema utilizzato estensivamente fa uso di **schermi a matrice attiva**: è più costoso, ma produce un'immagine di qualità superiore. In questo caso, invece di avere solamente due insiemi di fili paralleli, su uno degli elettrodi è presente, in corrispondenza di ciascun pixel, un piccolo elemento di commutazione. Accendendo o spegnendo questi elementi si può creare sullo schermo uno schema di tensioni, che corrisponde a un arbitrario pattern di bit. Gli elementi di commutazione sono chiamati **thin film transistor** ("transistor a pellicola sottile") e gli schermi piatti che li utilizzano sono spesso chiamati **schermi TFT**. Al giorno d'oggi la maggior parte dei calcolatori portatili e dei monitor a schermo piatto per computer desktop utilizza la tecnologia TFT.

Finora abbiamo descritto il funzionamento di uno schermo monocromatico. Anche se i dettagli sono molto più complicati, è sufficiente dire che gli schermi a colori si basano sugli stessi principi. Si usano filtri ottici per separare, in corrispondenza di ogni pixel, la luce bianca nelle componenti rossa, verde e blu, in modo da poterle visualizzare indipendentemente.

RAM della scheda video

Sia gli schermi CRT sia quelli TFT vengono ridisegnati dalle 60 alle 100 volte al secondo, accedendo a una memoria speciale, chiamata **RAM della scheda video** (*Video RAM*), che si trova sulla scheda del controllore dello schermo. Questa memoria ha una o più bitmap che rappresentano lo schermo; per esempio su uno schermo di 1600 × 1200 **pixel** (*picture element*, "elemento d'immagine") la RAM della scheda video deve contenere 1600 × 1200

valori, uno per ogni pixel. In realtà, per permettere un veloce passaggio da un'immagine dello schermo a un'altra, questa memoria potrebbe contenere più bitmap di questo tipo.

Su uno schermo di fascia alta ogni pixel dovrebbe essere rappresentato da un valore RGB (da *Red, Green* e *Blue*) composto da 3 byte, uno per l'intensità di ciascun componente (rosso, verde e blu) del colore del pixel. Le leggi dell'ottica ci dicono che qualsiasi colore può essere costruito combinando linearmente luci rosse, verdi e blu.

Una RAM della scheda video con 1600 × 1200 pixel e 3 byte per pixel richiede circa 5,5 MB per memorizzare l'immagine e una quantità non indifferente di tempo di CPU per operare su di essa. Per questa ragione alcuni calcolatori scendono al compromesso di utilizzare un unico numero a 8 bit per indicare il colore desiderato. Questo numero è utilizzato come indice di una tabella hardware, chiamata **tavolozza** (*color palette*), contenente 256 elementi, ciascuno dei quali memorizza un valore RGB a 24 bit. Questo tipo di strategia, chiamata **colore indicizzato**, riduce di 2/3 i requisiti della RAM della scheda video, ma permette di visualizzare su schermo solo 256 colori per volta. Spesso, quando sullo schermo sono mostrate contemporaneamente più finestre, solo i colori di una di queste vengono *renderizzati* correttamente; infatti, anche se di solito ogni finestra ha la sua propria mappatura, esiste una sola tavolozza hardware.

Gli schermi basati su bitmap richiedono una grande quantità di larghezza di banda. Per visualizzare dati multimediali a schermo intero occorre trasferire a ogni fotogramma 5,5 MB di dati. Nel caso di video in movimento (*full-motion*) sono necessari almeno 25 fotogrammi al secondo, che corrispondono a una velocità totale di trasferimento dati di 137,5 MB/s. Questo carico supera di molto le capacità di un bus (E)ISA ed è anche superiore a quanto era in grado di gestire il bus PCI originario (127,2 MB/s). Anche se immagini di minor risoluzione richiedono ovviamente un trasferimento di dati minore, la larghezza di banda rimane comunque un importante problema.

A partire dal Pentium II, per permettere una larghezza di banda maggiore tra le CPU e le RAM delle schede video, Intel ha cominciato a impiegare un nuovo bus verso la RAM video, chiamato **bus AGP** (*Accelerated Graphics Port*, "porta per la grafica veloce"), che può trasferire 32 bit a una frequenza di 66 MHz fornendo così una velocità di trasferimento dati di 252 MB/s. Sono state prodotte versioni successive del bus, funzionanti a velocità doppie, e maggiori 4 e 8 volte, per mettere a disposizione una larghezza di banda adatta alla grafica altamente interattiva senza sovraccaricare il bus PCI principale.

2.4.3 Mouse

Più il tempo passa, più i calcolatori sono utilizzati da persone con sempre minor conoscenza del loro funzionamento. I calcolatori della generazione ENIAC erano usati da coloro che li avevano costruiti, mentre negli anni '50 i computer venivano utilizzati soltanto da programmatori altamente specializzati. Oggigiorno invece i calcolatori sono largamente impiegati da persone che non sanno (o talvolta non vogliono sapere) troppi dettagli sul funzionamento dei calcolatori o sulla loro programmazione.

Agli inizi, la maggior parte dei calcolatori era dotata di un'interfaccia a linea di comando, attraverso la quale gli utenti impartivano i comandi. Dato che chi non è uno specialista dei calcolatori reputa che le interfacce a linea di comando siano spesso poco amichevoli, se non addirittura ostili, molti produttori di calcolatori, in seguito, hanno sviluppato interfacce punta-e-clicca, come quelle del Macintosh e di Windows. Per poter utilizzare questo tipo

d'interfaccia è necessario avere uno strumento per puntare sullo schermo, e il metodo più comune consiste nell'usare un mouse.

Il **mouse** è un piccolo oggetto di plastica situato a fianco della tastiera. Quando viene mosso sul tavolo, un piccolo puntatore si sposta, in corrispondenza, sullo schermo, permettendo agli utenti di puntare agli elementi desiderati. Il mouse, sulla parte superiore, ha uno, due o tre pulsanti che consentono all'utente di selezionare elementi e voci dei menu. Gli utenti più inesperti preferiscono i mouse con un unico tasto (è infatti difficile premere quello sbagliato se ce n'è soltanto uno), mentre quelli più abili preferiscono la potenza di un mouse a più tasti che permette azioni più sofisticate.

Sono stati prodotti tre tipi di mouse: meccanici, ottici e opto-meccanici. Il primo mouse aveva sul fondo due rotelle di gomma che sporgevano all'esterno; gli assi delle due rotelle erano disposti perpendicolarmente, di modo che, quando il mouse veniva spostato parallelamente a uno degli assi principali, ruotava soltanto una rotella. Ciascuna rotella guidava una resistenza variabile o un potenziometro, in modo che misurando il loro cambiamento era possibile calcolare la rotazione e, di conseguenza, la distanza percorsa dal mouse lungo una particolare direzione. Negli ultimi anni questo tipo di mouse è stato quasi completamente sostituito da un modello, mostrato nella Figura 2.33, che al posto delle due rotelle fa uso di una pallina leggermente sporgente sul fondo.

Il secondo tipo di mouse è quello ottico, che non ha né rotelle né pallina. Al loro posto ha, sul fondo, un **LED** (*Light Emitting Diode*, "diodo luminescente") e un fotorilevatore. Il mouse ottico viene utilizzato su uno speciale tappetino di plastica sul quale c'è una griglia di linee molto vicine fra loro; quando il mouse si muove il fotorilevatore riconosce l'incrocio tra due linee rilevando un cambiamento nella quantità di luce generata dal LED che viene riflessa. I circuiti all'interno del mouse contano il numero di linee attraversate lungo ogni direzione.

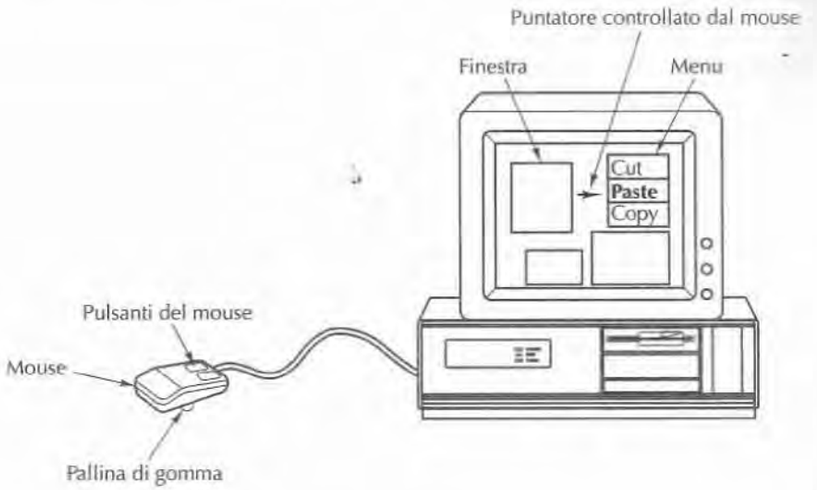


Figura 2.33 Mouse utilizzato per puntare a voci di un menu.

Il terzo tipo di mouse è quello opto-meccanico, anch'esso dotato di una pallina che fa ruotare due cilindretti perpendicolari tra loro. I cilindretti sono collegati a codificatori che hanno una serie di fori attraverso i quali può passare la luce; quando il mouse si sposta, i cilindretti ruotano e la luce colpisce il rilevatore ogni volta che un foro si trova allineato con il LED e il suo fotorilevatore. Il numero di impulsi che vengono rilevati è proporzionale allo spostamento effettuato.

Anche se sono possibili diverse soluzioni, in genere un mouse spedisce al calcolatore una sequenza di 3 byte ogni qualvolta si muove di una certa distanza minima (per esempio 0,01 pollici), chiamata in alcuni casi *mickey*. Di solito questi caratteri vengono spediti lungo una linea seriale, un bit alla volta. Il primo byte contiene un intero con segno che indica quante unità sono state percorse dal mouse nella direzione x rispetto all'ultima rilevazione, mentre il secondo byte fornisce la stessa informazione per quanto riguarda il movimento lungo y; il terzo byte contiene invece lo stato corrente dei pulsanti del mouse. A volte si utilizzano 2 byte per i valori di spostamento di ogni coordinata.

Il software del calcolatore riceve queste informazioni sui movimenti relativi del mouse, le converte in coordinate assolute e, in corrispondenza della posizione calcolata, visualizza una piccola freccia. Per selezionare un elemento sullo schermo l'utente deve soltanto posizionare la freccia sulla regione desiderata e premere un pulsante: il calcolatore può determinare quale elemento è stato selezionato dato che sa in ogni momento in che posizione dello schermo si trova la freccia.

2.4.4 Stampanti

In questo paragrafo descriveremo alcuni fra i più comuni tipi di stampanti monocromatiche (cioè in bianco e nero) e a colori.

Stampanti monocromatiche

Il tipo più economico di stampante è la **stampante a matrice di aghi**, in cui durante la stampa viene fatta scorrere lungo una linea una testina contenente dai 7 ai 24 aghi attivabili elettromagneticamente. Le stampanti a basso costo hanno sette aghi e possono stampare, per esempio, 80 caratteri per linea usando una matrice di 5 × 7 punti. In realtà una linea di stampa è composta da 7 linee orizzontali, ciascuna delle quali consiste di 5 × 80 = 400 punti. Ciascun punto può essere stampato oppure no, a seconda del carattere che deve essere rappresentato. La Figura 2.34(a) mostra come viene stampata la lettera "A" con una matrice 5 × 7.

È possibile migliorare la qualità di stampa utilizzando più aghi e facendo in modo che i punti stampati si sovrappongano parzialmente. La Figura 2.34(b) mostra una "A" stampata utilizzando 24 aghi che producono punti tra loro sovrapposti. Di solito sono necessarie più passate per ciascuna linea per poter sovrapporre i punti e quindi l'aumento della qualità va di pari passo con la diminuzione della velocità di stampa. La maggior parte delle stampanti a matrice può funzionare in modalità differenti che permettono di trovare un bilancio tra qualità di stampa e velocità.

Le stampanti ad aghi sono economiche (soprattutto in termini dei materiali di consumo) e molto affidabili, ma sono anche lente, rumorose e di bassa qualità. Nei sistemi attuali vengono impiegate in tre situazioni principali. Primo, sono molto diffuse per la stampa di grandi (> 30 cm) schede prestampate. Secondo, sono adatte a stampare su piccoli pezzi

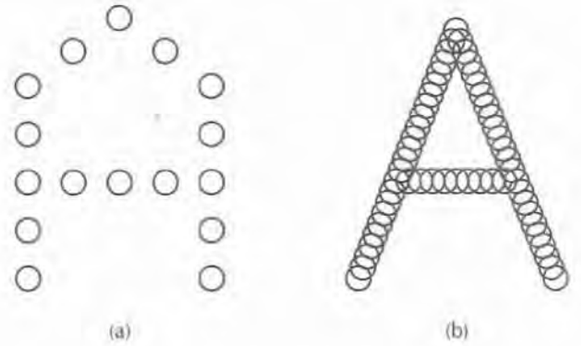


Figura 2.34 (a) Lettera "A" rappresentata con una matrice 5 × 7. (b) Lettera "A" stampata con 24 aghi in sovrapposizione.

di carta, come gli scontrini delle casse e dei bancomat, le ricevute delle carte di credito e le carte d'imbarco degli aerei. Terzo, sono di solito il mezzo meno costoso per stampare su modulo continuo più copie utilizzando la carta carbone.

Le **stampanti a getto d'inchiostro** sono il mezzo favorito per la stampa casalinga a basso costo. Una testina mobile, contenente una cartuccia d'inchiostro, scorre orizzontalmente lungo il foglio per mezzo di un nastro, mentre l'inchiostro viene spruzzato da piccoli ugelli. Le gocce d'inchiostro hanno un volume di circa un picolitro; ciò corrisponde a dire che una singola goccia d'acqua ne può contenere senza problemi anche 100 milioni.

Esistono due tipi di stampanti a getto d'inchiostro: piezoelettriche (usate da Epson) e termiche (usate da Canon, HP e Lexmark). Le stampanti piezolettriche a getto d'inchiostro hanno un particolare tipo di cristallo situato vicino al contenitore dell'inchiostro. Quando gli si applica una tensione, il cristallo si deforma leggermente, facendo uscire una gocciolina d'inchiostro dall'ugello. Il software può controllare la dimensione della goccia d'inchiostro, dato che risulta proporzionale alla tensione applicata.

Le stampanti a getto d'inchiostro termiche (chiamate anche stampanti *bubblejet*) contengono una piccola resistenza dentro ciascun ugello. Quando si applica loro una tensione la resistenza si scalda in modo estremamente veloce, facendo aumentare istantaneamente la temperatura dell'inchiostro fino al punto di ebollizione in modo da produrre, per vaporizzazione, una bolla di gas. La bolla di gas occupa più spazio dell'inchiostro che l'ha generata e produce quindi una pressione all'interno dell'ugello; l'inchiostro può muoversi soltanto verso l'apertura dell'ugello, da cui esce per finire sulla carta. Subito dopo l'ugello viene raffreddato e la differenza di pressione aspira all'interno della cartuccia un'altra goccia d'inchiostro. La velocità della stampante è legata alla rapidità con cui si può ripetere il ciclo di ebollizione/raffreddamento. Le gocce sono tutte della stessa dimensione, ma generalmente sono più piccole di quelle prodotte dalle stampanti piezolettriche.

Di solito le stampanti a getto d'inchiostro hanno una risoluzione di almeno 1200 **dpi** (*dots per inch*, "punti per pollice"), ma può anche essere quattro volte superiore. Sono economiche, silenziose e producono stampe di buona qualità, ma allo stesso tempo sono lente e utilizzano cartucce d'inchiostro costose. Quando si utilizzano le migliori stampanti a getto

d'inchiostro per stampare fotografie ad alta risoluzione su carta fotografica speciale il risultato è indistinguibile da una normale fotografia, anche per stampe di 8 × 10 pollici.

Dopo l'invenzione della stampa, da parte di Johann Gutenberg nel quindicesimo secolo, lo sviluppo più significativo nella riproduzione dei testi è probabilmente rappresentato dalla **stampante laser**. Questo dispositivo combina in un'unica periferica alta qualità dell'immagine, eccellente flessibilità, grande velocità e costo limitato. Le stampanti laser usano una tecnologia molto simile a quella delle fotocopiatrici; difatti molte società producono periferiche che permettono sia di effettuare fotocopie sia di stampare (e talvolta di spedire e ricevere fax).

La Figura 2.35 illustra la tecnologia di base. Il cuore della stampante è un tamburo rotante molto preciso (o, in alcuni sistemi di fascia alta, un nastro). All'inizio di ciascun ciclo di pagina il tamburo è caricato fino a circa 1000 volt e rivestito con un materiale fotosensibile. Successivamente la luce generata dal laser passa lungo tutta lunghezza del tamburo, come il fascio di elettroni in un CRT, con la differenza che, per ottenere la deviazione orizzontale si utilizza uno specchio ottagonale rotante invece che una tensione. Il fascio di luce viene modulato per produrre regioni luminose e regioni scure: le parti del tamburo colpite dal raggio perdono la loro carica elettrica.

Il tamburo, dopo aver disegnato una linea di punti, ruota di una frazione di grado per permettere il disegno della linea successiva. A questo punto la prima linea di punti raggiunge il *toner*, un contenitore di polvere nera elettrostaticamente sensibile. Il toner è attirato dai punti che hanno ancora una carica elettrica, formando così un'immagine visiva sulla linea del tamburo. Un istante dopo, ruotando, il tamburo ricoperto di *toner* viene premuto contro il foglio di carta, trasferendo su questo la polvere nera. Il foglio passa quindi attraverso dei rulli riscaldati per fissare l'immagine, fondendo in modo permanente il *toner* sulla carta. Nel seguito della rotazione il cilindro viene scaricato e ripulito di ogni residuo del *toner*, per

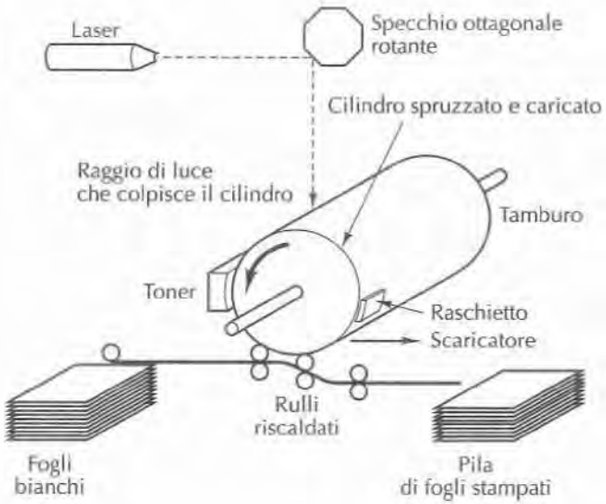


Figura 2.35 Funzionamento di una stampante laser.

poter essere nuovamente caricato e ricoperto di materiale fotosensibile, pronto per la stampa della pagina successiva.

È necessario precisare che questo processo è possibile grazie a una combinazione estremamente complessa di fisica, chimica, meccanica e ottica. Nonostante questa complessità alcuni produttori vendono dei meccanismi completi, chiamati **motori di stampa**, che comprendono tutte queste fasi. Per realizzare una stampante completa i produttori di stampanti laser combinano i motori di stampa con la propria elettronica e il proprio software. L'elettronica consiste in una veloce CPU integrata, alcuni megabyte di memoria per memorizzare la mappa dei bit di un'intera pagina e numerosi set di caratteri, alcuni dei quali predefiniti.

La maggior parte delle stampanti accetta comandi che descrivono come devono essere stampate le pagine. Questi comandi sono espressi in linguaggi come PCL di HP e PostScript di Adobe.

Le stampanti laser da 600 dpi e più possono stampare in modo accettabile fotografie in bianco e nero, ma la tecnologia è più complessa di quanto potrebbe sembrare. Si consideri una fotografia scannerizzata a 600 dpi che deve essere stampata con una stampante a 600 dpi. L'immagine scannerizzata contiene 600 × 600 pixel/pollice, ciascuno dei quali consiste in un valore di grigio compreso fra 0 (bianco) e 255 (nero). Anche la stampante può stampare a 600 dpi, ma ciascun pixel stampato è o nero (*toner* presente) o bianco (*toner* assente): i grigi non possono essere stampati.

Per riprodurre la scala dei grigi la tecnica più comune è quella dei **mezzitoni**, la stessa utilizzata per i poster commerciali. L'immagine è divisa in celle, solitamente di 6 × 6 pixel: ogni cella può quindi contenere tra 0 e 36 pixel neri. Le celle con più pixel sono percepite dall'occhio come più scure rispetto alle altre. I valori di grigio compresi tra 0 e 255 sono rappresentati dividendo questo intervallo in 37 zone. I valori da 0 a 6 appartengono alla zona 0, i valori da 7 a 13 alla zona 1 e così via (la zona 37 è leggermente più piccola delle altre in quanto 256 non è divisibile esattamente per 37). Ogni qualvolta si incontra un grigio appartenente alla zona 0, si lascia la sua cella dei mezzitoni bianca, come mostra la Figura 2.36(a). Un valore appartenente alla zona 1 è stampato con un pixel nero; uno della zona 2 con due pixel neri, come mostra la Figura 2.36(b). Le Figure 2.36(c)-(f) mostrano altre zone con valori diversi. Se si prende una fotografia scannerizzata a 600 dpi e la si rappresenta mediante questa tecnica, la sua risoluzione effettiva si riduce ovviamente a 100 celle/pollice; questa risoluzione viene chiamata **frequenza del retino dei mezzitoni**, ed è misurata convenzionalmente in **lpi** (*lines per inch*, "linee per pollice").

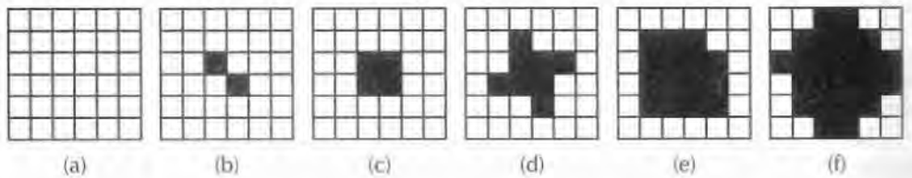


Figura 2.36 Mezzitoni per alcuni intervalli di livelli di grigio. (a) 0-6. (b) 14-20. (c) 28-34. (d) 56-62. (e) 105-111. (f) 161-167.

Stampanti a colori

Le immagini a colori vengono viste in due modi distinti: per luce trasmessa o per luce riflessa. Le immagini per luce riflessa, come quelle prodotte sui monitor CRT, sono create mediante la sovrapposizione di tre colori primari additivi, il rosso, il verde e il blu. Le immagini per luce riflessa, come le fotografie a colori e le immagini di riviste su carta lucida, assorbono alcune lunghezze d'onda di luce e riflettono le restanti. Queste immagini sono create dalla sovapposizione di tre colori primari sottrattivi, il ciano (completamente assorbito dal rosso), il giallo (completamente assorbito dal blu) e il magenta (completamente assorbito dal verde). In teoria ogni colore può essere riprodotto mischiando i colori ciano, giallo e magenta, ma in pratica è difficile produrre inchiostri sufficientemente puri da ottenere un vero nero. Per questo motivo quasi tutti i sistemi di stampa a colori utilizzano quattro colori: il ciano, il giallo, il magenta e il nero, e sono chiamati **stampanti CMYK** (dalle iniziali inglesi dei quattro colori, con K che sta per blacK, nero). Per rappresentare i colori i monitor usano, al contrario, luce trasmessa e il sistema RGB.

L'insieme completo di colori rappresentabili da uno schermo o da una stampante è chiamato **gamut**, cioè **gamma dei colori.** Nessun dispositivo ha una gamma di colori che corrisponde a quelli del mondo reale, dato che ogni colore può al massimo assumere 256 intensità, che producono in totale solo 16.777.216 colori discreti. A causa di alcune imperfezioni dovute alla tecnologia, il numero totale di colori si riduce ulteriormente e quelli rimanenti non sono neanche distribuiti in modo uniforme nello spettro dei colori. Inoltre la percezione dei colori non dipende solo dalle leggi della fisica, ma è determinata in buona parte anche dal funzionamento dei coni e dei bastoncelli dell'occhio umano.

Di conseguenza si vede che non è affatto banale convertire un'immagine a colori che appare correttamente sullo schermo in un'identica immagine stampata. I problemi principali che si incontrano in questo processo sono i seguenti:

1. i monitor a colori usano luce trasmessa, mentre le stampanti a colori usano luce riflessa
2. i CRT generano 256 intensità per colore, mentre le stampanti usano i mezzitoni
3. i monitor hanno un fondo nero, mentre la carta ha un colore chiaro
4. le gamme dei colori RGB e CMYK sono diverse.

Per ottenere immagini stampate che corrispondano alla realtà (e anche alle immagini a schermo) è necessario calibrare i dispositivi, utilizzare software sofisticati e affidarsi a una considerevole esperienza da parte dell'utente.

Le cinque tecnologie di stampa attualmente in uso si basano sul sistema CMYK. A livello più basso troviamo le stampanti a colori a getto d'inchiostro. Esse funzionano come quelle monocromatiche, anche se hanno quattro cartucce (per i colori C, M, Y e K) al posto di una sola; forniscono a un basso prezzo risultati di buona qualità per la grafica e risultati accettabili per le fotografie (le stampanti sono economiche, ma non le cartucce).

Per ottenere risultati migliori bisognerebbe utilizzare carte e inchiostri speciali. Esistono due tipi d'inchiostro: **a base di coloranti** e **a pigmenti**. I primi consistono in coloranti diluiti in un liquido, producono colori luminosi e fluiscono facilmente, ma il loro principale svantaggio è quello di sbiadirsi quando sono esposti a radiazioni ultraviolette, come quelle della luce solare. I secondi contengono invece delle particelle solide di pigmento sospese in un liquido che evapora dalla carta depositando il colore. Questi inchiostri non si sbiadiscono con il tempo, ma non sono luminosi quanto quelli basati su coloranti e le particelle di pigmento hanno la tendenza a bloccare gli ugelli, rendendone necessaria una periodica pulizia. Per stampare fotografie è necessario usare un tipo di carta lucida o patinata appositamente progettata per trattenere le gocce d'inchiostro e impedire che si spandano eccessivamente.

Un gradino sopra le stampanti a getto d'inchiostro troviamo le **stampanti a inchiostro solido** che contengono quattro speciali inchiostri a cera solidificati in blocchi che vengono successivamente sciolti all'interno di serbatoi riscaldati. Per queste stampanti il tempo di attesa all'accensione può raggiungere i 10 minuti, per consentire lo scioglimento dei blocchi d'inchiostro. L'inchiostro caldo viene spruzzato sulla carta, dove si solidifica e si fonde con essa grazie all'utilizzo di due rulli rigidi.

Il terzo tipo di stampante a colori è quella laser. Funziona come la sua "cugina" monocromatica, tranne per il fatto che vengono definite quattro diverse immagini C, M, Y e K trasferite su un rullo usando quattro toner distinti. Dato che l'intera bitmap deve essere generata prima della stampa, un'immagine da 1200 × 1200 dpi per una pagina di 80 pollici quadrati richiede 115 × 10^6 pixel. Per memorizzare la bitmap nella stampante, utilizzando per esempio 4 bit/pixel, sono necessari 55 MB, senza considerare la memoria richiesta dai processori interni, dal repertorio dei caratteri, ecc. Per questo motivo le stampanti laser a colori sono costose; la loro velocità di stampa è però elevata, la qualità è alta e i colori delle immagini sono stabili nel tempo.

Il quarto tipo di stampante a colori è la **stampante a getto di cera**. Essa ha un ampio nastro rivestito di quattro inchiostri a cera e suddiviso in settori lunghi quanto la larghezza della pagina. Quando la carta passa sotto il nastro migliaia di elementi riscaldanti sciolgono la cera, che si fonde con la carta sotto forma di pixel utilizzando il sistema CMYK. Una volta le stampanti a cera erano la principale tecnologia di stampa a colori, ma ora stanno per essere sostituite da altri tipi i cui materiali di consumo sono più economici.

L'ultimo tipo di stampante a colori è la **stampante a sublimazione**. Anche se il termine ha un che di freudiano, fisicamente indica il passaggio dallo stato solido a quello gassoso, senza passare da quello liquido. Un noto materiale che sublima è il ghiaccio secco (diossido di carbonio ghiacciato), un altro è lo zolfo. In una stampante a sublimazione un elemento mobile contenente i coloranti CMYK passa sopra una testina di stampa in cui vi sono migliaia di elementi riscaldanti programmabili; i coloranti vengono vaporizzati e assorbiti da una carta speciale collocata vicino alla testina. Ciascun elemento riscaldante può produrre 256 livelli di temperatura; maggiore è la temperatura, maggiore è la quantità di colorante che si deposita e quindi l'intensità del colore. Diversamente da tutte le altre stampanti a colori non è necessario ricorrere alla tecnica dei mezzitoni, dato che è possibile creare una gamma quasi continua di colori. Di solito le stampati fotografiche di piccole dimensioni usano il processo di sublimazione per creare immagini fotografiche altamente realistiche su carta speciale (e costosa).

2.4.5 Apparecchiature per le telecomunicazioni

Al giorno d'oggi la maggior parte dei calcolatori è connessa a una rete di calcolatori, che spesso è Internet. In questo paragrafo descriveremo come funzionano le apparecchiature che permettono l'accesso a queste reti.

Modem

Dato che l'uso dei calcolatori si è diffuso fortemente negli ultimi anni, è sempre più comune che due computer debbano comunicare fra loro. Molti, per esempio, hanno un PC a casa e lo utilizzano per comunicare con il loro calcolatore al lavoro, con un Internet provider oppure con un sistema bancario on-line. In molti casi è la linea telefonica a permettere la comunicazione fisica tra due sistemi.

Una linea telefonica non è adatta a trasmettere i segnali di un calcolatore, che di solito rappresentano il bit 0 con 0 volt e il bit 1 con 3 o 5 volt, come mostra la Figura 2.37(a). Quando sono trasmessi su una linea telefonica progettata per la voce, i segnali a due livelli subiscono una notevole distorsione, provocando di conseguenza errori di trasmissione. Tuttavia un segnale dalla forma sinusoidale e con una frequenza che varia tra 1000 e 2000 Hz può essere trasmesso con una distorsione relativamente piccola; l'uso di questo segnale, chiamato **portante**, è alla base della maggior parte dei sistemi di telecomunicazione.

Dato che le pulsazioni di un'onda sinusoidale sono conosciute a priori, una sinusoide perfetta non trasmette alcuna informazione. Variandone però l'ampiezza, la frequenza o la fase è possibile trasmettere una sequenza di 1 o di 0, come mostra la Figura 2.37. Questo processo è conosciuto con il termine di **modulazione**; nella **modulazione d'ampiezza** [vedi Figura 2.37(b)] si utilizzano due diverse tensioni per rappresentare rispettivamente i valori 0 e 1. Se si ascoltasse la trasmissione di dati digitali a una velocità molto bassa si sentirebbe un forte rumore per i valori 1 e nessun rumore per i valori 0.

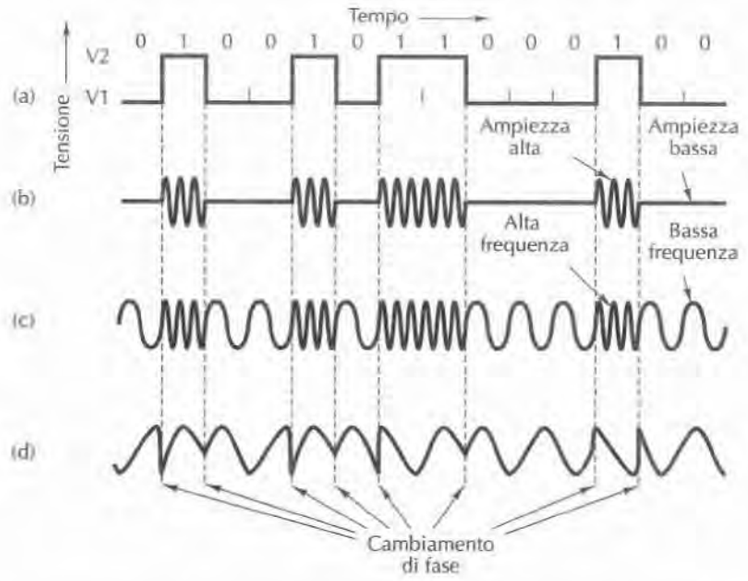


Figura 2.37 Trasmissione bit a bit della stringa 01001011000100 su una linea telefonica. (a) Segnale a due livelli. (b) Modulazione d'ampiezza. (c) Modulazione di frequenza. (d) Modulazione di fase.

Nella **modulazione di frequenza** [vedi Figura 2.37(c)] la tensione rimane a un livello costante, ma la frequenza della portante cambia in corrispondenza dei valori 1 e 0. Se si ascoltassero i dati digitali modulati rispetto alla frequenza si sentirebbero due toni distinti, in corrispondenza dei valori 0 e 1. La modulazione di frequenza è spesso indicata con il termine **modulazione (numerica) di frequenza** (*frequency shift keying*).

Nella **modulazione di fase** [vedi Figura 2.37(d)] l'ampiezza e la frequenza non cambiano, ma la fase della portante è invertita di 180 gradi quando i dati passano da 0 a 1 o viceversa. In sistemi a modulazione di fase più sofisticati la fase della portante cambia improvvisamente di 45, 135, 225 o 315 gradi all'inizio di ogni intervallo di tempo, in modo da poter rappresentare 2 bit per ogni intervallo; questa codifica viene chiamata **modulazione a coppia di bit**. Uno sfasamento della fase di 45 gradi potrebbe rappresentare per esempio 00, uno di 135 gradi potrebbe rappresentare 01, e così via. Esistono anche altri schemi che permettono di trasmettere 3 o più bit per intervallo. Il numero di intervalli di tempo (cioè il numero di potenziali cambiamenti del segnale al secondo) è chiamato **baud**. Se si utilizzano 2 o più bit per intervallo, il valore della velocità in bit sarà maggiore di quella in baud; è un errore molto comune confondere queste due unità di misura.

Se i dati da trasmettere consistono in una serie di caratteri a 8 bit, potrebbe essere comodo disporre di una connessione in grado di trasmettere simultaneamente 8 bit. Per ottenere ciò servirebbero 8 coppie di fili, ma, dato che le linee telefoniche forniscono un solo canale, i bit devono essere spediti in modo seriale, uno dopo l'altro (o in gruppi di due se si usa la modulazione a coppia di bit). Con il termine modem si indica il dispositivo che riceve da un calcolatore i caratteri sotto forma di segnali a due livelli, un bit alla volta, e che trasmette i bit in gruppi di uno o due utilizzando una modulazione di ampiezza, di frequenza o di fase. Di solito si fa precedere ogni carattere a 8 bit da un bit d'inizio e lo si fa seguire da un bit di fine, in modo da marcarne le estremità; in questo modo ogni carattere richiede 10 bit.

Un modem in fase di trasmissione spedisce i singoli bit di un carattere a intervalli di tempo regolari. Una velocità di 9600 baud significa per esempio che il segnale cambia ogni 104 μs. Si utilizza poi un modem in ricezione per convertire una portante modulata in stringhe binarie. Dato che i bit giungono al ricevente a intervalli di tempo uniformi, il modem determina l'inizio del carattere e poi sincronizza il suo orologio per sapere quando deve campionare la linea per leggere i bit in entrata.

I modem moderni funzionano a una tasso di trasferimento dati compreso tra 28.800 e 57.600 bit/s, e a un tasso in baud generalmente molto più basso. Per poter spedire più bit per baud, si usa una combinazione di tecniche diverse, modulando ampiezza, frequenza e fase. Quasi tutti questi modem sono **full-duplex**, il che significa che possono trasmettere allo stesso tempo in entrambe le direzioni (utilizzando frequenze diverse). Al contrario i modem e le linee di trasmissione che in un dato istante possono trasmettere in una sola direzione sono chiamate **half-duplex**. Le linee che possono trasmettere in una sola direzione sono invece chiamate **simplex**.

Digital subscriber line

Quando le società telefoniche riuscirono a raggiungere la velocità di 56 Kbps, si complimentarono tra loro con grande entusiasmo. Nel frattempo però l'industria della TV via cavo offriva già velocità superiori a 10 Mbps su cavi condivisi e la TV satellitare stava rag-

giungendo sistemi da 50 Mbps. Dato che l'accesso a Internet aveva assunto un'importanza sempre maggiore nei loro affari, le cosiddette **telcos** (da *telephone companies*, compagnie telefoniche) cominciarono a rendersi conto che avevano bisogno di un prodotto più competitivo rispetto alle linee *dial-up*. La loro risposta fu l'offerta di un nuovo servizio di accesso digitale a Internet; i servizi che offrono una larghezza di banda maggiore rispetto al servizio telefonico standard sono spesso chiamati a **banda larga** (*broadband*), anche se in realtà il termine non ha un vero significato tecnico, ma è più che altro utilizzato a fini promozionali.

All'inizio vi erano molte offerte che si sovrapponevano e che rientravano, al variare di *x*, sotto il nome generale di **xDSL** (*Digital Subscriber Line*). In seguito descriveremo il servizio **ADSL** (*Asymmetric DSL*, DSL asimmetrica), destinato con ogni probabilità a diventare il più diffuso. Dato che lo sviluppo della ADSL è ancora in corso e non sono stati definiti tutti gli standard, alcuni dei dettagli seguenti potrebbero cambiare in futuro; ciononostante l'immagine che ne verrà data dovrebbe rimanere sostanzialmente valida. Per maggiori informazioni riguardo la ADSL si veda (Summers, 1999 e Vetter et al., 2000).

La lentezza dei modem deriva dal fatto che i telefoni sono stati inventati per trasportare la voce umana e l'intero sistema è stato attentamente ottimizzato a questo scopo e i dati hanno sempre interpretato il ruolo dei "fratellastri". Il cavo che collega un abbonato alla compagnia telefonica, chiamato **ciclo locale**, di solito viene limitato a 3000 Hz. Questo vincolo, che limita anche la velocità di trasferimento dati, viene ottenuto mediante un filtro che si trova nell'ufficio della compagnia telefonica. La larghezza di banda reale del ciclo locale dipende dalla sua lunghezza e può raggiungere 1,1 MHz per distanze di pochi chilometri.

La Figura 2.38 mostra l'approccio più comune nell'offerta del servizio ADSL. Quello che in pratica viene fatto consiste nel suddividere il ciclo locale in 256 canali indipendenti di 4312,5 Hz ciascuno. Il canale 0 è usato per il servizio telefonico tradizionale, **POTS** (*Plain Old Telephone Service*). I canali 1-5 non sono utilizzati e servono a mantenere separati la voce e i dati in modo che non interferiscano tra loro. Dei 250 canali rimanenti, uno è usato per il controllo del traffico in uscita, un altro per il controllo del traffico in entrata, mentre gli altri sono a disposizione per la trasmissione dei dati. Una ADSL corrisponde quindi a 250 modem.

In teoria ciascuno dei canali rimanenti può essere utilizzato per un flusso dati full-duplex, ma in pratica le armoniche, le interferenze e altri effetti mantengono i sistemi reali molto al di sotto di questo dato teorico. Spetta al fornitore di servizi Internet determinare quanti canali sono usati in uscita e quanti in entrata. Dato che la maggior parte degli utenti

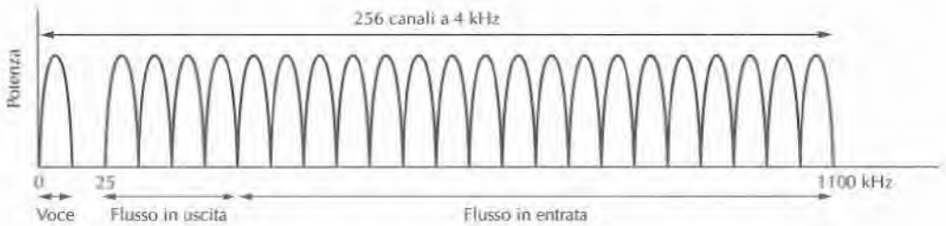


Figura 2.38 Funzionamento di una ADSL.

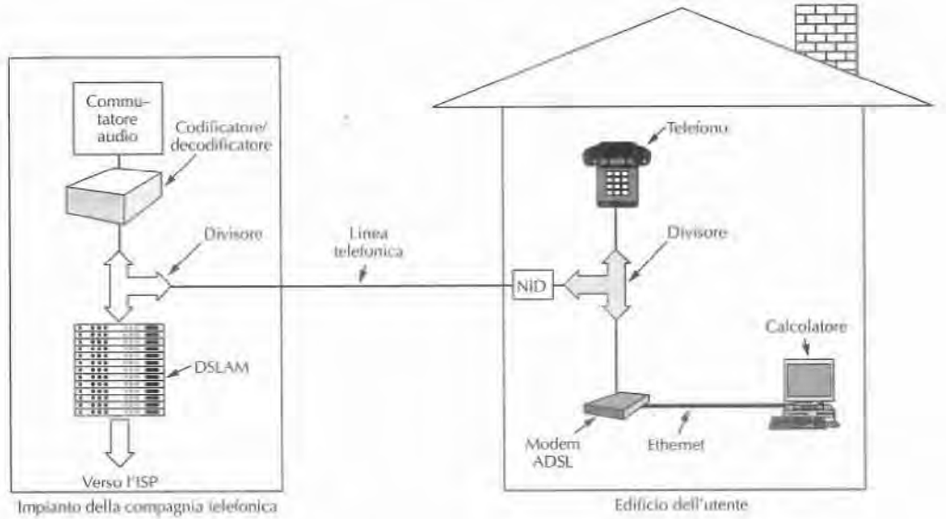


Figura 2.39 Tipica configurazione dei dispositivi per la ADSL.

scarica più dati di quanti non ne trasmetta, la maggior parte dei fornitori di servizi alloca l'80%-90% della larghezza di banda al canale in entrata (anche se una suddivisione 50-50 tra i due canali sarebbe tecnicamente possibile). A questa scelta si deve la lettera "A" nell'acronimo ADSL. Una suddivisione piuttosto comune prevede di utilizzare 32 canali in uscita e gli altri in entrata.

All'interno di ogni canale si controlla costantemente la qualità della linea e, se necessario, si modifica la velocità di trasferimento dati: per questo motivo canali differenti possono avere velocità diverse. I dati vengono spediti con una combinazione di modulazione di ampiezza e di fase, che permette di trasmettere fino a 15 bit per baud. Per esempio, con 224 canali in entrata e 15 bit/baud a 4000 baud, si ottiene una larghezza di banda in entrata di 13,44 Mbps. In pratica il rapporto tra segnale e rumore non è mai abbastanza buono per ottenere questa velocità, ma su cicli ad alta qualità e su distanze brevi è possibile raggiungere velocità di 8 Mbps.

La Figura 2.39 mostra una tipica organizzazione di una ADSL. In questo schema l'utente o un tecnico della compagnia telefonica devono installare un dispositivo d'interfaccia, **NID** (*Network Interface Device*) presso l'edificio del cliente. Questa piccola scatola di plastica segna la fine della proprietà della compagnia telefonica e l'inizio di quella dell'utente. Vicino al NID (o talvolta combinato con quest'ultimo) è presente un **divisore** (*splitter*), cioè un filtro analogico che divide la banda 0-4000 Hz (per il servizio telefonico tradizionale) dai dati. Il segnale POTS è instradato verso il telefono o il fax esistenti, mentre il segnale dei dati è instradato verso un modem ADSL. Il modem ADSL è un processore di segnale digitale impostato appositamente per funzionare come 250 modem in parallelo a diverse frequenze. Dato che la maggior parte dei modem ADSL attuali è esterna, il calcolatore deve essere connesso attraverso un collegamento ad alta velocità. Di solito ciò si ottiene inserendo una scheda Ethernet (la più diffusa tra le reti locali) nel calcolatore e creando una piccola

Ethernet a due nodi che comprende soltanto il calcolatore e il modem ADSL. A volte si utilizza una porta USB al posto della Ethernet, ma in futuro saranno senza dubbio disponibili schede specifiche per modem ADSL.

All'altra estremità del cavo, presso la compagnia telefonica, è installato un altro divisore. Qui la parte del segnale relativa alla voce è separata tramite un filtro e spedita verso un normale commutatore per la voce. I segnali a frequenza maggiore di 26 kHz sono instradati verso un nuovo tipo di dispositivo, chiamato **DSLAM** (*Digital Subscriber Line Access Multiplexer*), che contiene lo stesso tipo di processore di segnale digitale all'interno del modem ADSL. Dopo che il segnale digitale è stato riconvertito in un flusso di bit, si creano i pacchetti di dati da spedire all'ISP.

Internet via cavo

Oggi molte società di TV via cavo offrono l'accesso a Internet attraverso i propri cavi. Dato che la tecnologia differisce significativamente da quella della ADSL, vale la pena analizzarla brevemente. In ogni grande città c'è una sede principale dell'operatore via cavo, mentre, sparse sul territorio, ci sono un gran numero di scatole piene di componenti elettronici, chiamate **stazioni di testa**. Le stazioni di testa sono connesse alla sede principale da fibre ottiche o da cavi con un'alta larghezza di banda.

Da ciascuna stazione di testa partono dei cavi che raggiungono centinaia di case e uffici e i clienti si collegano al cavo che passa vicino alla propria abitazione. Centinaia di utenti condividono quindi uno stesso cavo che parte dalla stazione di testa e la cui larghezza di banda, in genere, è di circa 750 MHz. Questo sistema è radicalmente differente dalla ADSL dove gli utenti hanno un cavo privato (cioè non condiviso) che li connette alla sede della compagnia telefonica. Tuttavia, all'atto pratico, non vi è una gran differenza tra avere il proprio canale a 1,1 MHz verso la sede della compagnia telefonica e dover condividere lo spettro di 200 MHz di un cavo connesso a una stazione di testa con altri 400 utenti, metà dei quali non lo usa nello stesso istante. Ciò significa però che un utente di Internet via cavo avrà un servizio migliore alle 4 di mattina, piuttosto che alle 4 di pomeriggio, mentre le prestazioni del servizio ADSL sono costanti per tutto il giorno. Chi voglia avere un servizio Internet via cavo dalle ottime prestazioni dovrebbe considerare l'ipotesi di trasferirsi in un quartiere residenziale (gli edifici sono lontani gli uni dagli altri e quindi ci sono pochi clienti per cavo) oppure in un quartiere povero (nessuno può permettersi un servizio Internet).

Il fatto che il cavo sia un mezzo condiviso pone il grande problema di determinare chi può spedire dati, quando e a quale frequenza. Per vedere come funziona tutto ciò, occorre prima descrivere brevemente come funziona la TV via cavo. Nel Nord America i canali televisivi via cavo occupano di solito l'intervallo tra 50 e 550 MHz (tranne per le frequenze delle radio FM comprese tra 88 e 108 MHz). Questi canali hanno una larghezza di 6 MHz, comprese le bande di sicurezza necessarie per prevenire la sovrapposizione tra canali contigui. In Europa lo schema di allocazione è simile, anche se la banda parte da 65 MHz e ogni canale occupa tra i 6 e gli 8 MHz, per via della maggiore risoluzione richiesta dai sistemi PAL e SECAM. La parte inferiore della banda non è utilizzata per la trasmissione televisiva.

Per aggiungere il servizio di Internet, le società via cavo devono risolvere due problemi:

1. come aggiungere l'accesso a Internet senza interferire con i programmi TV
2. come ottenere un traffico bidirezionale, dato che gli amplificatori sono per loro natura unidirezionali.

Le soluzioni adottate sono le seguenti. I cavi moderni funzionano bene anche oltre i 750 MHz. I canali in uscita (cioè dall'utente alla stazione di testa) sono collocati nella banda compresa tra 5 e 42 MHz (leggermente più alta in Europa), mentre il traffico in entrata (cioè dalla stazione di testa all'utente) utilizza la parte alta della banda totale (Figura 2.40).

Occorre notare che, dato che i segnali televisivi sono tutti in entrata, è possibile utilizzare amplificatori in uscita solo nella regione tra 5 e 42 MHz e amplificatori in entrata che lavorano dai 54 MHz in su, com'è mostrato nella figura. Di conseguenza si ottiene un'asimmetria tra le bande in entrata e quelle in uscita, dato che è disponibile una regione di spettro sopra le frequenza dedicata alla televisione maggiore che non al di sotto. D'altra parte il traffico è principalmente in entrata e quindi ciò non impensierisce né rattrista gli operatori via cavo. D'altra parte, come abbiamo già visto, anche le compagnie telefoniche forniscono spesso un servizio di DSL asimmetrico, senza che ve ne sia alcuna esigenza tecnica.

L'accesso a Internet richiede un apposito modem, che abbia due interfacce: una verso il calcolatore e l'altra verso il cavo della rete. L'interfaccia tra il calcolatore e il modem via cavo è semplice e di solito si tratta di una scheda Ethernet, come nel caso della ADSL. In futuro l'intero modem potrebbe essere una piccola scheda inserita all'interno del calcolatore, allo stesso modo dei modem interni V.9x.

L'altra interfaccia è invece più complicata. Una gran parte dello standard via cavo è legata all'ingegneria delle trasmissioni radio, un argomento che esula dallo scopo di questo libro. Il solo aspetto che vale la pena sottolineare è che i modem via cavo, come quelli ADSL, sono sempre attivi: stabiliscono una connessione non appena la macchina viene accesa e la mantengono finché non viene spenta, dato che le tariffe degli operatori via cavo non dipendono dal tempo di connessione.

Per comprendere meglio il loro funzionamento, vediamo che cosa succede quando un modem via cavo viene collegato e acceso. Il modem effettua una scansione dei canali in entrata per cercare uno speciale pacchetto spedito periodicamente dalla stazione di testa e che fornisce alcuni parametri di sistema. Non appena trova tale pacchetto il modem comunica la propria presenza utilizzando uno dei canali in uscita. A questo punto la stazione di testa risponde assegnando al modem alcuni canali in entrata e in uscita; la stazione di testa può modificare in un secondo momento tale assegnamento per bilanciare il carico di traffico quando lo reputa necessario.

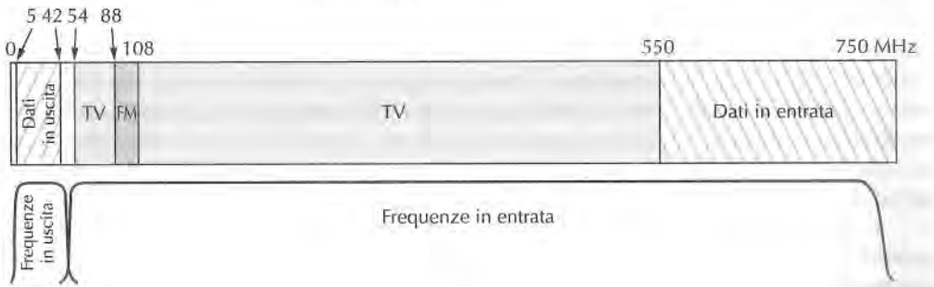


Figura 2.40 Allocazione delle frequenze in un classico sistema di TV via cavo utilizzato per l'accesso a Internet.

Il modem determina quindi la propria distanza dalla stazione di testa spedendo uno speciale pacchetto e calcolando quanto tempo impieghi per ricevere una risposta; questo processo è chiamato **ranging**. È importante che il modem conosca questa distanza affinché possa regolare appropriatamente l'utilizzo di canali in uscita e per impostarne correttamente la temporizzazione. Questi canali sono divisi in intervalli temporali chiamati **minislot**; ogni pacchetto di dati in uscita deve entrare in uno o più *minislot* consecutivi. La stazione di testa comunica periodicamente l'inizio di una nuova serie di *minislot*, ma, per via dei tempi di propagazione sul cavo, questo segnale non è sentito simultaneamente da tutti i modem. Conoscendo la distanza che lo separa dalla stazione di testa ogni modem può calcolare quanto tempo prima era stato in realtà spedito il primo minislot. La lunghezza dei *minislot* dipende dalla rete, ma una tipica lunghezza del campo dati è di 8 byte.

Durante l'inizializzazione la stazione di testa assegna a ciascun modem un *minislot* da usare per richiedere una parte della larghezza di banda in uscita. In genere uno stesso *minislot* viene assegnato a più modem, generando problemi di contesa. Quando un calcolatore vuole spedire un pacchetto, lo trasferisce al modem, che richiede il numero di *minislot* necessario; se la stazione di testa accetta la richiesta, spedisce una conferma sul canale in entrata indicando al modem quali *minislot* sono stati riservati per il suo pacchetto. Il pacchetto viene quindi spedito a partire dal *minislot* che gli è stato allocato ed è inoltre possibile utilizzare un campo dell'intestazione per richiedere pacchetti aggiuntivi.

Se invece si verifica una contesa per la richiesta dei *minislot*, non verrà spedita alcuna conferma e il modem aspetterà un tempo casuale prima di riprovare. Dopo ciascun fallimento, il tempo casuale di attesa viene raddoppiato di modo che il carico si distribuisca meglio nel tempo quando il traffico è molto intenso.

I canali in entrata sono gestiti in maniera differente da quelli in uscita. Per prima cosa, esiste un solo mittente (la stazione di testa) e quindi non ci può essere contesa, e non c'è bisogno dei *minislot*, che in realtà non sono altro che una multiplazione statistica a divisone di tempo. In secondo luogo, poiché il traffico in entrata è solitamente molto più grande di quello in uscita, si utilizza una dimensione di 204 byte per tutti i pacchetti. Una parte di questi byte contengono un codice a correzione di errore Reed-Solomon e altre informazioni di servizio, lasciando solo 184 byte ai dati dell'utente. Questi valori sono stati scelti per essere compatibili con la televisione terrestre basata su MPEG-2, in modo che la formattazione dei canali della TV e di quelli del traffico in entrata sia la stessa. La Figura 2.41 mostra l'organizzazione logica delle connessioni.

Tornando all'inizializzazione del modem, dopo aver terminato l'operazione di *ranging* e ottenuto i propri canali in uscita, quelli in entrata e l'assegnamento dei *minislot*, il modem è libero di iniziare a spedire pacchetti. Questi giungono alla stazione di testa, che li trasmette via cavo su un canale dedicato alla sede principale della compagnia e quindi al fornitore di servizi (che potrebbe essere la compagnia stessa). Il primo pacchetto serve a richiedere all'ISP un indirizzo di rete (tecnicamente un indirizzo IP), assegnato dinamicamente. Il modem inoltre richiede, e ottiene, l'ora del giorno con grande precisione.

Il passo successivo riguarda la sicurezza. Dato che i cavi sono condivisi, chi lo volesse potrebbe leggere tutto il traffico che passa da lui. Per questo motivo il traffico viene interamente criptato in entrambe le direzioni, per evitare che qualcuno ficchi il naso (letteralmente) nei dati dei propri vicini. Una parte della procedura d'inizializzazione serve a stabilire le chiavi di cifratura. A prima vista si potrebbe pensare che è impossibile che due estranei, la stazione

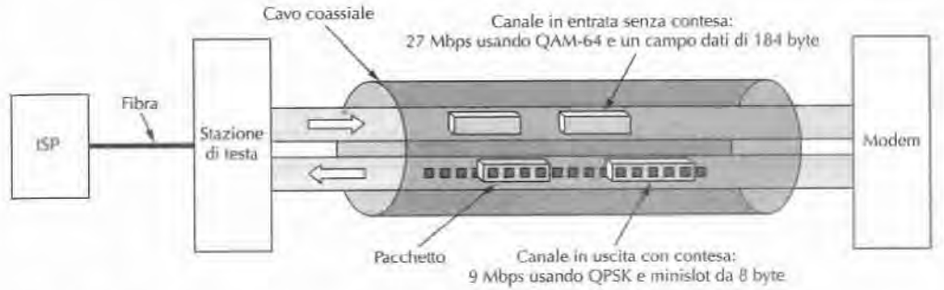


Figura 2.41 Tipici dettagli dei canali in entrata e in uscita nel Nord America.

di testa e il modem, possano stabilire una chiave segreta alla luce del sole e davanti a migliaia di persone che li osservano. In realtà ciò è possibile, ma la tecnica utilizzata (l'algoritmo Diffie-Hellman) va oltre lo scopo di questo libro. Per un'analisi al riguardo si veda (Kaufman et al. 2002).

Infine il modem deve farsi identificare e fornire il proprio numero identificativo univoco sul canale sicuro. A questo punto l'inizializzazione è completa e l'utente può registrarsi presso l'ISP e cominciare a utilizzare il servizio.

Ci sarebbero molte altre cose da dire a proposito dei modem via cavo; alcuni significativi riferimenti bibliografici sono (Adams e Dulchinos, 2001; Donaldson e Jones, 2001 e Dutta-Roy, 2001).

2.4.6 Macchine fotografiche digitali

Le macchine fotografiche digitali sono ormai diventate una periferica del calcolatore, dato che i computer vengono sempre più frequentemente usati anche per la fotografia digitale. Spieghiamo brevemente come funziona. Tutte le macchine fotografiche hanno una lente che forma nella parte posteriore, internamente all'apparecchio, un'immagine del soggetto inquadrato. In una macchina fotografica di tipo classico la parte posteriore è coperta da una pellicola su cui si forma un'immagine latente nel momento in cui viene colpita dalla luce; durante lo sviluppo delle foto si può rendere visibile questa immagine utilizzando dei composti chimici. Una macchina fotografica digitale funziona nello stesso modo tranne per il fatto che al posto della pellicola c'è una griglia rettangolare di **CCD** (*Charge-Coupled Device*, "dispositivo ad accoppiamento di carica"), sensibili alla luce (alcune macchine fotografiche digitali usano i CMOS, ma noi ci concentreremo sui più comuni CCD).

Quando un CCD è colpito dalla luce si carica elettricamente in modo proporzionale alla quantità di luce ricevuta. Questa carica può essere letta da un convertitore analogico-digitale e trasformata in un valore compreso tra 0 e 255 (per gli apparecchi economici) oppure tra 0 e 4095 (per le macchine fotografiche reflex digitali con un'unica lente). Il funzionamento è schematizzato nella Figura 2.42.

Ciascun CCD produce un unico valore, indipendentemente dal colore della luce che lo colpisce. Per formare un'immagine a colori i CCD sono organizzati in gruppi di quattro elementi. Un **filtro di Bayer** è posizionato sopra il CCD per permettere che la luce rossa colpisca solo uno dei quattro CCD in ciascun gruppo, la luce blu ne colpisca soltanto un

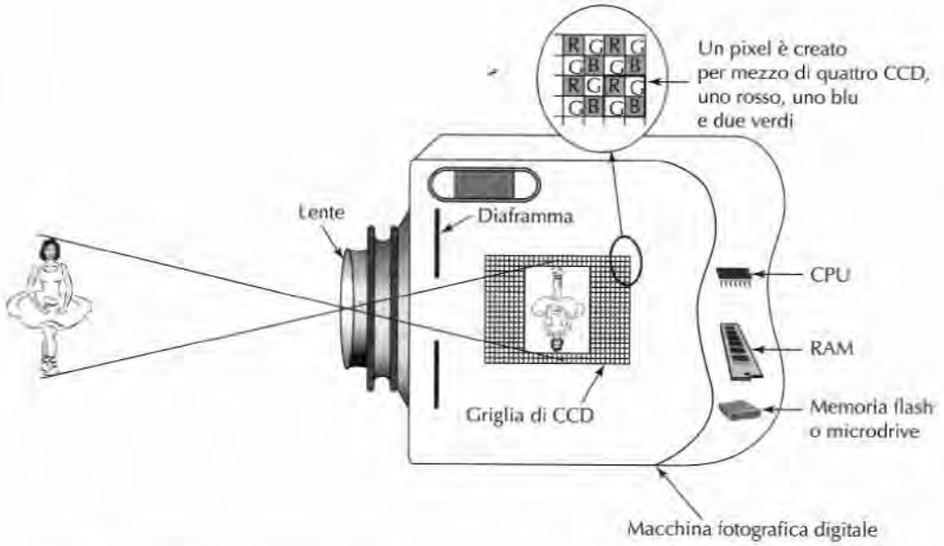


Figura 2.42 Macchina fotografica digitale.

altro e la luce verde i due restanti. Si utilizzano due elementi per il verde, in quanto è molto più pratico utilizzare quattro CCD per ogni pixel al posto di tre e perché l'occhio umano è più sensibile alla luce verde che alle luci rosse e blu. Quando un produttore di macchine fotografiche digitali afferma che un modello ha, per esempio, 6 milioni di pixel, sta mentendo. La macchina ha 6 milioni di CCD, che corrispondono a 1,5 milioni di pixel. L'immagine sarà letta come una griglia di 2828 × 2121 pixel (sulle macchine a basso costo) oppure di 3000 × 2000 pixel (sulle SLR digitali), mentre i pixel aggiuntivi sono prodotti per interpolazione dal software interno alla macchina fotografica.

Quando viene premuto il pulsante per scattare una foto il software compie tre azioni: imposta la messa a fuoco, determina l'esposizione e calcola il bilanciamento del bianco. La messa a fuoco automatica funziona analizzando le alte frequenze dell'immagine e muovendo la lente finché non vengono massimizzate al fine di ottenere il maggiore dettaglio possibile. L'esposizione è determinata misurando la luce che raggiunge i CCD e regolando di conseguenza il diaframma della lente e il tempo di esposizione, di modo che l'intensità luminosa corrisponda al punto medio dell'intervallo gestito dai CCD. Il bilanciamento del bianco consiste nel misurare lo spettro della luce incidente e nell'effettuare, successivamente, le necessarie correzioni cromatiche.

Successivamente l'immagine viene letta dai CCD e memorizzata come una griglia di pixel nella memoria interna della macchina digitale. Le macchine digitali SLR di livello professionale, utilizzate dai fotografi professionisti, sono in grado di scattare otto fotogrammi ad alta risoluzione al secondo per 5 secondi; esse richiedono circa 1 GB di RAM interna per salvare le immagini prima di poterle elaborare e memorizzare in modo permanente. Le macchine fotografiche meno costose hanno una quantità di memoria RAM leggermente minore.

Nella fase successiva alla cattura dell'immagine, il software della macchina effettua la correzione per il bilanciamento del bianco, in modo da compensare le componenti rossastre o bluastre della luce (dovute per esempio a un soggetto in ombra o all'uso del flash). In seguito applica un algoritmo per la riduzione del rumore e un altro per compensare i CCD difettosi. Tenta poi di rendere l'immagine più definita e meno sfocata (a meno che questa funzione non sia stata disabilitata) cercando i contorni e aumentando l'intensità del gradiente intorno a loro.

Infine l'immagine può essere compressa per ridurre la quantità di spazio richiesta. Un formato comunemente utilizzato è **JPEG** (*Joint Photographic Experts Group*), che prevede l'applicazione della trasformata di Fourier bidimensionale e il taglio di alcune componenti ad alta frequenza. Questa trasformazione riduce il numero di bit necessari a rappresentare l'immagine, sacrificando però i dettagli più fini.

Quando sono terminate tutte le elaborazioni interne alla macchina fotografica, l'immagine viene memorizzata, generalmente su una memoria flash o un hard disk rimovibile di piccole dimensioni chiamato ***microdrive***. La fase di post-elaborazione e la scrittura possono richiedere alcuni secondi per ciascuna immagine.

Quando l'utente torna a casa può collegare la macchina fotografica digitale al calcolatore utilizzando di solito un cavo USB o FireWire. Le immagini vengono così trasferite dalla macchina fotografica al disco del calcolatore. Utilizzando programmi appositi, come Adobe Photoshop, l'utente può ritagliare l'immagine, regolarne luminosità, contrasto, bilanciamento cromatico, nitidezza e fuoco, rimuovere parti dell'immagine e applicare numerosi filtri. Quando è soddisfatto del risultato l'utente può stampare i file delle immagini con una stampante a colori, spedirle via Internet allo stampatore oppure può riversarle su un CD-ROM o un DVD per archiviarle ed eventualmente stamparle in un secondo momento. In una macchina fotografica SLR la quantità di potenza computazionale, di RAM, di spazio su hard disk e di software necessario è notevole. Il calcolatore non solo deve compiere tutte le azioni descritte, ma anche comunicare in tempo reale con la CPU della lente e con la CPU del flash, ridisegnare l'immagine sullo schermo LCD e gestire tutti i pulsanti, rotelle, luci, display e tasti della macchina. Si tratta di un sistema integrato estremamente potente, che spesso, in termini di prestazioni, è paragonabile ai calcolatori desktop di qualche anno fa.

2.4.7 Codifica dei caratteri

Ogni calcolatore ha un insieme di caratteri che, come minimo indispensabile, comprende le 26 lettere maiuscole, le 26 lettere minuscole, le cifre da 0 a 9 e un insieme di simboli speciali, come spazio, punto, virgola, segno meno e ritorno a capo.

Per poter utilizzare questi caratteri nel calcolatore occorre assegnare loro un numero: per esempio a = 1, b = 2, …, z = 26, + = 27, − = 28. La corrispondenza tra caratteri e numeri naturali costituisce un **codice di caratteri**. È necessario che due calcolatori che comunicano fra loro utilizzino lo stesso codice, altrimenti non saranno in grado di capirsi. Per questa ragione sono stati definiti alcuni standard, di cui esamineremo i due più importanti.

ASCII

Un codice ampiamente utilizzato si chiama **ASCII** (*American Standard Code for Information Interchange*, "standard americano per lo scambio d'informazioni"). I caratteri ASCII sono definiti da 7 bit, permettendo così un totale di 128 caratteri distinti. La Figura 2.43 mostra

Esa	Nome	Significato	Esa	Nome	Significato
0	NUL	Nullo	10	DLE	Uscita trasmissione (Data Link Escape)
1	SOH	Inizio intestazione (Start Of Heading)	11	DC1	Controllo periferica 1
2	STX	Inizio testo (Start Of Text)	12	DC2	Controllo periferica 2
3	ETX	Fine testo (End Of Text)	13	DC3	Controllo periferica 3
4	EOT	Fine trasmissione (End Of Transmission)	14	DC4	Controllo periferica 4
5	ENQ	Interrogazione (Enquiry)	15	NAK	Riconsocimento negativo (Negative AcKnowledgement)
6	ACK	Riconoscimento (ACKnowledgement)	16	SYN	Annulla (SYNchronous Idle)
7	BEL	Campanello (BELL)	17	ETB	End of Transmission Block
8	BS	BackSpace	18	CAN	CANcel
9	HT	Tabulazione orizzontale (Horizontal Tab)	19	EM	Fine supporto (End of Medium)
A	LF	Riga nuova (Line Feed)	1A	SUB	Sostituisci (SUBstitute)
B	VT	Tabulazione verticale (Vertical Tab)	1B	ESC	Esc (ESCape)
C	FF	Avanzamento carta/nuova pagina (Form Feed)	1C	FS	Separatore di file (File Separator)
D	CR	Ritorno a capo (Carriage Return)	1D	GS	Separatore di gruppi (Group Separator)
E	SO	Disinserzione (Shift Out)	1E	RS	Separatore di record (Record Separator)
F	SI	Inserzione (Shift In)	1F	US	Separatore di unità (Unit Separator)

Esa	Car	Esa	Car	Esa	Car	Esa	Car	Esa	Car	Esa	Car
20	Spazio	30	0	40	@	50	P	60	`	70	p
21	!	31	1	41	A	51	Q	61	a	71	q
22	"	32	2	42	B	52	R	62	b	72	r
23	#	33	3	43	C	53	S	63	c	73	s
24	$	34	4	44	D	54	T	64	d	74	t
25	%	35	5	45	E	55	U	65	e	75	u
26	&	36	6	46	F	56	V	66	f	76	v
27	'	37	7	47	G	57	W	67	g	77	w
28	(	38	8	48	H	58	X	68	h	78	x
29	)	39	9	49	I	59	Y	69	i	79	y
2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
2C	,	3C	<	4C	L	5C	\	6C	l	7C	|
2D	-	3D	=	4D	M	5D	]	6D	m	7D	}
2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
2F	/	3F	?	4F	O	5F	_	6F	o	7F	DEL

Figura 2.43 Caratteri ASCII.

il codice ASCII. I codici compresi tra 0 e 1F (in esadecimale) sono caratteri di controllo e non vengono stampati.

Molti dei caratteri di controllo ASCII sono pensati per la trasmissione di dati. Un messaggio potrebbe per esempio consistere di un carattere SOH (*Start of Header*, "inizio intestazione"), un'intestazione, un carattere STX (*Start of Text*, "inizio testo"), il testo stesso, un ETX (*End of Text*, "fine testo") e infine un carattere EOT (*End of Transmission*, "fine trasmissione"). In realtà i messaggi spediti sulle linee telefoniche e sulle reti sono formattati in maniera leggermente differente e i caratteri di controllo ASCII pensati per la trasmissione non sono praticamente più utilizzati.

I caratteri ASCII stampabili comprendono lettere maiuscole e minuscole, cifre, simboli di punteggiatura e alcuni simboli matematici.

UNICODE

Il mercato dei calcolatori è nato e cresciuto principalmente negli Stati Uniti, il che ha portato a un insieme di caratteri, quello ASCII, adatto alla lingua inglese, ma meno per altre lingue. I francesi hanno bisogno degli accenti (per esempio, *système*), i tedeschi dei segni diacritici (per esempio, *für*), e così via. Alcune lingue europee hanno delle lettere che non sono comprese nel codice ASCII, come la tedesca ß e la danese ø. Altre hanno alfabeti completamente differenti (per esempio, il russo o l'arabo) e alcune lingue addirittura non hanno alfabeto (per esempio, il cinese). Dato che i computer si sono diffusi nei quattro angoli del mondo e i rivenditori di software desiderano vendere i propri prodotti anche nei paesi in cui la maggior parte della gente non parla inglese, è sorta la necessità di un diverso insieme di caratteri.

Il primo tentativo di espandere il codice ASCII fu il codice IS 646, che aggiungeva ulteriori 128 caratteri, trasformandolo così in un codice a 8 bit chiamato **Latin-1**. I caratteri aggiuntivi erano per lo più lettere latine accentate o con segni diacritici. Il tentativo successivo fu il codice IS 8859 che introdusse il concetto di **code page**, un insieme di 256 caratteri specifico di una particolare lingua o di un gruppo di lingue. Il codice IS 8859-1 corrisponde a Latin-1, il codice IS 8859-2 gestisce alcune lingue slave (per esempio, il ceco e il polacco), mentre il codice IS 8859-3 contiene i caratteri richiesti dal turco, dal maltese, dall'esperanto, dal galiziano e da altre lingue. Il problema principale del *code page* è che il software deve tener traccia di quale pagina è attiva, rendendo tra l'altro impossibile mischiare lingue diverse in uno stesso documento; inoltre questo schema non copre né il giapponese né il cinese.

Un gruppo di produttori di calcolatori ha deciso di risolvere il problema formando un consorzio per creare un nuovo sistema, chiamato **UNICODE**, che successivamente è diventato uno Standard Internazionale (IS 10646).

Oggi UNICODE è supportato da alcuni linguaggi di programmazione (per esempio, Java), da alcuni sistemi operativi (per esempio, Windows XP) e da molte applicazioni. Dato che il mercato dei calcolatori tende a diventare sempre più globale è molto probabile che questo standard venga accettato sempre di più.

L'idea alla base di UNICODE consiste nell'assegnare a ogni carattere un valore a 16 bit, chiamato **code point**; non esistono quindi caratteri o sequenze speciali composte da più byte. Il fatto che ogni simbolo sia composto da 16 bit rende più semplice la scrittura di programmi.

Utilizzando simboli a 16 bit, UNICODE ha 65.536 *code point*. Dato che le lingue di tutto il mondo usano complessivamente circa 200.000 simboli, i *code point* sono una risorsa scarsa che deve essere allocata con molta attenzione. Circa metà dei code point sono già stati assegnati e il consorzio UNICODE valuta in continuazione nuove proposte per i rimanenti. Per far sì che il codice UNICODE fosse accettato più velocemente, il consorzio ha deciso intelligentemente di utilizzare Latin-1 per i *code point* compresi tra 0 e 255, rendendo più facile la conversione tra ASCII e UNICODE. Inoltre, per evitare uno spreco eccessivo di *code point*, ogni segno diacritico ha il suo proprio *code point* e spetta al software combinarlo con la lettera vicina per formare un nuovo carattere.

Lo spazio dei *code point* è diviso in blocchi, multipli di 16. All'interno di UNICODE a ciascun alfabeto principale è assegnata una sequenza di zone consecutive; alcuni esempi (e il numero di *code point* allocati) sono il latino (336), il greco (144), il cirillico (256), l'armeno (96), l'ebraico (112), il devanagari (128), il gurmukhi (128), l'oriya (128), il telugu (128) e il kannada (128). Si noti che a queste lingue sono stati assegnati più *code point* delle loro lettere. Questa scelta deriva in parte dal fatto che molte lingue hanno più forme per ciascuna lettera; ciascuna lettera inglese ha, per esempio, due forme: quella minuscola e quella MAIUSCOLA. Alcune lingue hanno tre o quattro forme, talvolta dipendenti dalla posizione della lettera all'interno di una parola, se si trova cioè all'inizio, nel mezzo o alla fine.

I *code point*, oltre alle lettere di questi alfabeti, sono stati assegnati anche ai segni diacritici (112), ai simboli di punteggiatura (112), ai caratteri soprascritti e sottoscritti (48), ai simboli matematici (256), alle forme geometriche (96) e ai simboli ornamentali (192).

Dopo questi *code point* ci sono i simboli richiesti dal cinese, dal giapponese e dal coreano. Prima ci sono 1024 simboli fonetici (cioè il katakana e il bopomofo) e poi gli ideogrammi Han (20.992), usati sia dal cinese sia dal giapponese, e infine le sillabe coreane Hangul (11.156).

6400 *code point* sono inoltre stati allocati per uso locale, di modo che gli utenti possano definire caratteri speciali per usi particolari.

Anche se UNICODE risolve molti problemi legati all'internazionalizzazione, esso non risolve (e non cerca di risolvere) tutti i problemi globali. Per esempio, mentre le lettere dell'alfabeto latino sono nel loro ordine corretto, gli ideogrammi Han non sono ordinati come nel dizionario. Da ciò ne consegue che un programma inglese può ordinare alfabeticamente le parole *cat* e *dog* controllando i valori UNICODE delle loro iniziali, mentre lo stesso non può però essere fatto con un programma giapponese, che necessita di tabelle esterne per capire quale simbolo preceda l'altro.

Un altro problema è che con il tempo continuano ad apparire nuove parole. Cinquant'anni fa nessuno parlava di *applet, cyberspazio, gigabyte, laser, modem, smiley o videocassette*. Se in inglese l'aggiunta di nuove parole non richiede nuovi *code point*, in giapponese sì. Oltre a nuove parole tecniche, c'è la richiesta di aggiungere almeno 20.000 nuovi nomi personali (soprattutto cinesi) e di luoghi. I non vedenti ritengono che il linguaggio Braille dovrebbe far parte di UNICODE, così come altri gruppi con particolari interessi (di tutti i tipi) vorrebbero che ciò che percepiscono fosse codificato, di loro diritto, in *code point*. Il consorzio UNICODE analizza e valuta tutte le nuove proposte.

UNICODE utilizza lo stesso *code point* per i caratteri che, in giapponese e cinese, hanno un aspetto simile, ma un significato differente o una scrittura leggermente diversa (come se gli elaboratori di testo inglesi scrivessero "*blue*" come "*blew*" solo perché suonano allo stesso modo). Alcune persone interpretano questo fatto come un'ottimizzazione dovuta alla scarsità di *code point*; altri come un esempio di imperialismo culturale anglosassone (assegnare ai caratteri valori a 16 bit non era forse una scelta altamente politica?). Per rendere la cosa ancora peggiore si pensi che un dizionario giapponese completo contiene 50.000 kanji (nomi esclusi); avendo a disposizione soltanto 20.992 *code point* per gli ideogrammi Han, è necessario compiere delle scelte. Non tutti i giapponesi ritengono che un consorzio di produttori di calcolatori, anche se alcuni sono giapponesi, sia il foro ideale per prendere queste decisioni.

2.5 Riepilogo

I sistemi di calcolo sono costituiti da tre tipi di componenti: processori, memorie e dispositivi di I/O. Il compito di un processore è quello di prelevare istruzioni, decodificarle ed eseguirle. Il ciclo preleva-decodifica-esegui può sempre essere descritto come un algoritmo e, difatti, è spesso eseguito da un interprete software che funziona a un livello inferiore. Per guadagnare velocità molti calcolatori hanno una o più pipeline oppure un'architettura superscalare con più unità funzionali che lavorano in parallelo.

I sistemi con più processori sono sempre più diffusi. I calcolatori paralleli comprendono gli array di processori, nei quali la stessa operazione è calcolata su più insiemi di dati allo stesso tempo, i multiprocessori, nei quali più CPU condividono una stessa memoria, e i multicomputer, nei quali più calcolatori, ciascuno dotato di una propria memoria, comunicano scambiandosi messaggi.

Le memorie possono essere classificate come primarie o secondarie. La memoria primaria è utilizzata per contenere il programma che viene eseguito in un dato momento. Il suo tempo di accesso è breve, di poche decine di nanosecondi al massimo, e indipendente dall'indirizzo al quale si vuole accedere. Le cache riducono ulteriormente questo tempo di accesso. Alcune memorie sono inoltre equipaggiate con codici a correzione d'errore per migliorarne l'affidabilità.

Le memorie secondarie, al contrario, hanno tempi di accesso molto più lunghi (millisecondi e più) e dipendenti dalla posizione del dato che deve essere letto o scritto. Le più comuni memorie secondarie sono i nastri, i dischi magnetici e i dischi ottici. Esistono diversi tipi di dischi magnetici, tra i quali i floppy disk, i Winchester, gli IDE, gli SCSI e i RAID. I dischi ottici comprendono i CD-ROM, i CD-R e i DVD.

I dispositivi di I/O sono utilizzati per trasferire informazioni dal e nel calcolatore, e sono collegati al processore e alla memoria mediante uno o più bus. Alcuni esempi comprendono i terminali, i mouse, le stampanti e i modem. La maggior parte dei dispositivi di I/O utilizza il codice di caratteri ASCII, anche se UNICODE sta rapidamente guadagnando sempre più consenso man mano che il mondo dei computer si va globalizzando.

Problemi

1. Consideriamo il funzionamento di una macchina con il percorso dati mostrato nella Figura 2.2. Supponiamo che il caricamento dei registri di input richieda 5 ns, l'esecuzione della ALU 10 ns e la memorizzazione del risultato nel registro 5 ns. Qual è il numero massimo di MIPS che può raggiungere questa macchina senza far ricorso alla pipeline?
2. Qual è lo scopo del passo 2 nella lista del Paragrafo 2.1.2? Che cosa succederebbe se questo passo fosse omesso?
3. Sul calcolatore 1 tutte le istruzioni richiedono 10 ns per essere eseguite. Sul calcolatore 2 esse richiedono invece 5 ns. Si può dire con certezza che il calcolatore 2 è più veloce? Si argomenti la risposta.
4. Immaginiamo di progettare un calcolatore a singolo chip per un sistema integrato. Il chip conterrà anche tutta la memoria e funzionerà alla stessa velocità della CPU senza alcuna penalizzazione per l'accesso ai dati. Si esaminino i principi illustrati nel Paragrafo 2.1.4 e si dica se essi sono così importanti (ipotizzando che si desideri ottenere alte prestazioni).
5. Un certo calcolo è altamente sequenziale, in quanto ogni passo dipende dai precedenti. Per questa elaborazione sarebbe più appropriato un array di processori o un processore a pipeline? Si spieghi la scelta.
6. Per competere con le macchine tipografiche appena inventate, un monastero medioevale decise di riprodurre libri scritti a mano in un gran numero di copie facendo lavorare insieme, in un'enorme stanza, numerosi amanuensi. L'abate pronunciava ad alta voce la prima parola del libro da riprodurre e tutti gli amanuensi la trascrivevano. L'abate pronunciava quindi la seconda parola e gli amanuensi la trascrivevano. Questo processo veniva ripetuto finché l'intero libro non era letto ad alta voce e copiato. Quale fra i sistemi con parallelismo a livello di processore descritti nel Paragrafo 2.1.6 assomiglia più da vicino a questo sistema?
7. Scendendo nei cinque livelli della gerarchia di memorie trattati nel testo, i tempi di accesso aumentano. Si faccia una ragionevole stima del rapporto del tempo di accesso dei dischi ottici rispetto a quello dei registri della memoria. Si assuma che i dischi siano sempre caricati nel lettore.
8. Alla tipica domanda: "Credi al topolino dei dentini caduti?" i sociologi possono fornire tre risposte, sì, no e non so. In base a ciò la Società di Calcolatori Sociomagnetici ha deciso di costruire un calcolatore per elaborare i dati dei sondaggi. Questo computer ha una memoria trinaria, cioè ciascun byte (tryte?) consiste di 8 *trit*, ognuno dei quali può contenere i valori 0, 1 oppure 2. Quanti trit sono necesari per memorizzare un numero a 6 bit? Si dia un'espressione per il numero di trit necessari per memorizzare n bit.
9. Si calcoli la velocità di trasferimento dati dell'occhio umano utilizzando le seguenti informazioni. Il campo visivo è composto da circa 10^6 elementi (pixel). Ciascun pixel può essere ridotto alla sovrapposizione di tre colori primari, ciascuno dei quali ha 64 intensità. La risoluzione temporale è di 100 ms.
10. Si calcoli la velocità di trasferimento dati dell'orecchio umano usando le seguenti informazioni. Le persone possono sentire frequenze fino a 22 KHz. Per catturare tutta l'informazione contenuta in un segnale sonoro a 22 KHz è necessario campionare il suono al doppio della frequenza, cioè a 44 KHz. Con ogni probabilità un campione a 16 bit è sufficiente per catturare la maggior parte delle informazioni uditive (cioè l'orecchio non può distinguere più di 65.536 livelli d'intensità).

11. In tutti gli esseri viventi l'informazione genetica è codificata nelle molecole di DNA. Una molecola di DNA è una sequenza lineare dei quattro nucleotidi base: *A*, *C*, *G* e *T*. Il genoma umano contiene approssimativamente 3×10^9 nucleotidi sotto forma di circa 30.000 geni. Qual è la capacità totale (in bit) dell'informazione del genoma umano? Qual è la massima capacità d'informazione (in bit) del gene medio?
12. Un certo calcolatore può essere equipaggiato con 268.435.456 byte di memoria. Perché un produttore dovrebbe scegliere un numero così particolare, invece di un numero più facile da ricordare come 250.000.000?
13. Si definisca un codice di Hamming a 7 bit con parità pari per le cifre comprese tra 0 e 9.
14. Si definisca un codice per le cifre comprese tra 0 e 9 la cui distanza di Hamming sia 2.
15. In un codice di Hamming alcuni bit vengono "sprecati", nel senso che sono utilizzati per effettuare il controllo e non per memorizzare l'informazione. Qual è la percentuale di bit sprecati per messaggi la cui lunghezza totale (dati + bit di controllo) è $2^n - 1$? Si valuti numericamente questa espressione per valori di n compresi tra 3 e 10.
16. Il disco mostrato nella Figura 2.19 ha 1024 settori per traccia e una velocità di rotazione di 7200 RPM. Qual è la velocità di trasferimento dati del disco per una singola traccia?
17. Un calcolatore ha un bus con un tempo di ciclo di 5 ns, durante il quale può leggere o scrivere dalla memoria una parola a 32 bit. Il calcolatore ha un disco Ultra4-SCSI che utilizza un bus e funziona a 160 MB/s. Solitamente la CPU preleva ed esegue un'istruzione a 32 bit a ogni intervallo di 1 ns. In che misura il disco rallenta la CPU?
18. Si immagini di dover scrivere la parte di un sistema operativo dedicata alla gestione del disco. Dal il punto di vista logico il disco sarà rappresentato come una sequenza di blocchi, a partire da 0 nella parte interna fino a un certo valore massimo nella parte esterna. Quando i file vengono creati occorre allocare settori liberi, cosa che si può fare a partire dall'interno oppure dall'esterno. Ha importanza la scelta della strategia? Si motivi la risposta.
19. Quanto tempo è necessario per leggere un disco con 10.000 cilindri, ciascuno dei quali contiene quattro tracce di 2048 settori? Prima devono essere letti tutti i settori della traccia 0 a partire dal settore 0, poi tutti i settori della traccia 1 a partire dal settore 0, e così via. Il tempo di rotazione è di 10 ms e una ricerca richiede 1 ms tra cilindri adiacenti e 20 ms nel caso peggiore. Lo spostamento da una traccia all'altra dello stesso cilindro può essere fatta istantaneamente.
20. Il RAID livello 3 è in grado di correggere errori singoli utilizzando solo un disco di parità. Qual è la particolarità del RAID livello 2? Dopo tutto anch'esso è in grado di correggere solo gli errori singoli, ma necessita di un numero maggiore di dischi.
21. Qual è l'esatta capacità (in byte) di un CD-ROM Modo 2 per il supporto, ormai standardizzato, da 80 minuti? Qual è la capacità disponibile per i dati degli utenti utilizzando il Modo 1?
22. Per masterizzare un CD-R il laser deve pulsare ad alta frequenza. Qual è la lunghezza di un impulso, in nanosecondi, quando la frequenza è 10x e si utilizza il Modo 1?
23. Per poter inserire 133 minuti di video su un DVD a singolo lato e singolo strato è necessario utilizzare un'elevata compressione. Si calcoli il fattore di compressione richiesto, assumendo che sia disponibile uno spazio di 3,5 GB per la traccia video, che la risoluzione dell'immagine sia di 720 × 480 pixel con colori a 24 bit e che le immagini siano visualizzate a 30 frame/s.
24. I Blu-Ray hanno una velocità di 4,5 MB/s e una capacità di 25 GB. Quanto tempo è necessario per leggere l'intero disco?

25. La velocità di trasferimento dati tra la CPU e la sua memoria associata è di vari ordini di grandezza maggiore rispetto alla velocità delle componenti meccaniche di I/O. In che modo questo sbilanciamento può causare inefficienze? Com'è possibile alleviarle?

26. Un produttore afferma che il proprio terminale può visualizzare una bitmap con 2^{24} colori diversi. Se l'hardware ha 1 solo byte per ciascun pixel, com'è possibile ottenere questo valore?

27. Un terminale ha uno schermo con una risoluzione di 1600 × 1200, che viene ridisegnato 75 volte al secondo. Quanto è lungo l'impulso che corrisponde a un pixel?

28. Una stampante monocromatica può stampare in ogni pagina 50 linee da 80 caratteri. Un carattere occupa in media un riquadro di 4 mm^2, di cui circa il 25% è ricoperto da *toner*, mentre il resto ne è privo. Uno strato di *toner* ha uno spessore di 25 μ e una cartuccia di *toner* per la stampante misura 25 × 8 × 2 cm. Per quante pagine può essere utilizzata una cartuccia?

29. Quando un testo ASCII con parità dispari viene trasmesso in modo asincrono alla velocità di 5600 caratteri/s su un modem da 56.000 bps, qual è la percentuale dei bit ricevuti che contiene effettivamente dati (e non è overhead)?

30. La società di Modem Hi-Fi ha appena progettato un nuovo modem a modulazione di frequenza che usa 64 frequenze invece di solo 2. Ciascun secondo è diviso in n intervalli temporali uguali, ognuno dei quali contiene uno dei 64 possibili toni. Quanti bit per secondo può trasmettere questo modem, utilizzando la trasmissione asincrona?

31. Un utente Internet si è abbonato a un servizio ADSL a 2 Mbps. La sua vicina si è abbonata a un servizio di Internet via cavo che utilizza una banda condivisa di 12 MHz. Lo schema di modulazione utilizzato è QAM-64 e ci sono n case per cavo, ciascuna con un calcolatore. Solo una frazione f di questi calcolatori è in linea nello stesso momento. In quali condizioni l'utente via cavo ottiene un servizio migliore dell'utente ADSL?

32. Una macchina fotografica digitale ha una risoluzione di 3000 × 2000 pixel e usa 3 byte/pixel per rappresentare i colori RGB. Il produttore della macchina fotografica vorrebbe poter scrivere su una memoria flash, in soli 2 s, un'immagine JPEG con un fattore di compressione 5x. Quale velocità di trasferimento dati è richiesta?

33. Una macchina fotografica digitale professionale ha un sensore con 16 milioni di pixel, ciascuno con 3 byte/pixel. Quante immagini possono essere memorizzate su una memoria flash da 1 GB se il fattore di compressione è 5x? Si assuma che 1 GB valga 2^{30} byte.

34. Si stimi quanti caratteri, spazi compresi, contiene in genere un libro di informatica. Quanti bit sono necessari per codificare un libro usando il codice ASCII con parità? Quanti CD-ROM servono per immagazzinare una libreria di 10.000 libri? Quanti DVD a doppio lato e doppio strato sono necessari per memorizzare la stessa libreria?

35. Si scriva una procedura *hamming* (*ascii, codificata*) che converta i 7 bit meno significativi di *ascii* in una parola di codice che sia un intero a 11 bit e che venga memorizzato in *codificata*.

36. Si scriva una funzione *distanza* (*codice, n, k*) che prenda come input un array *codice* di n caratteri di k bit ciascuno e restituisca come output la distanza dell'insieme di caratteri.

Capitolo 3

Livello logico digitale

Alla base della gerarchia mostrata nella Figura 1.2 si trova il livello logico digitale, ovvero il vero e proprio hardware del calcolatore. In questo capitolo approfondiremo quegli aspetti della logica digitale che serviranno poi da base per lo studio dei livelli superiori. Anche se gli argomenti affrontati si trovano al confine tra l'informatica e l'elettronica il materiale è "autocontenuto": per poterne seguire la trattazione non sono quindi necessarie specifiche conoscenze ingegneristiche o relative all'hardware.

Gli elementi base a partire dai quali sono costruiti tutti i calcolatori digitali sono incredibilmente semplici. Inizieremo il nostro studio partendo da questi componenti elementari e dalla particolare algebra a due valori (algebra di Boole) che si utilizza per analizzarli. Successivamente esamineremo alcuni circuiti fondamentali, tra cui quelli per eseguire calcoli aritmetici costruiti mediante semplici combinazioni di porte logiche. L'argomento successivo sarà l'organizzazione della memoria, ovvero com'è possibile combinare le porte logiche per memorizzare informazioni. In seguito verrà affrontato il tema delle CPU e in particolare il modo in cui CPU composte da un unico chip si interfacciano con memoria e periferiche. Nella parte finale del capitolo saranno presentati, a titolo di esempio, numerosi prodotti reali.

3.1 Porte logiche e algebra di Boole

I circuiti digitali possono essere costruiti combinando tra loro un piccolo numero di componenti elementari. Nei paragrafi successivi descriveremo questi elementi base, mostreremo come possono essere combinati e introdurremo una potente tecnica matematica utilizzabile per analizzarne il comportamento.

3.1.1 Porte logiche

Un circuito digitale è un circuito in cui sono presenti solo due valori logici[1]. Generalmente un valore (per esempio il numero binario 0) è rappresentato da un segnale compreso tra 0 e 1 volt, mentre l'altro valore (per esempio il numero binario 1) è rappresentato da un se-

[1] In perfetta analogia a quanto accade nella logica classica (*N.d.Rev.*).

gnale compreso tra 2 e 5 volt; le tensioni al di fuori di questi intervalli non sono ammesse. La base hardware di tutti i calcolatori digitali è costituita da alcuni piccoli dispositivi elettronici, chiamati **porte logiche** (*gate*), ciascuna delle quali calcola una diversa funzione di questi segnali.

I dettagli sul funzionamento interno delle porte logiche esulano dallo scopo di questo libro, in quanto rientrano nel campo del **livello dei dispositivi**, che si trova al di sotto del nostro livello 0. Ciononostante, faremo ora una piccola digressione al riguardo, senza però entrare nei dettagli più complessi, in modo da fornire un'immagine d'insieme dell'idea base. Tutta la moderna logica digitale si fonda, in ultima analisi, sul fatto che un transistor può essere costruito in modo da funzionare come un velocissimo interruttore binario. Nella Figura 3.1(a) è possibile vedere un transistor integrato in un semplice circuito. I transistor hanno tre connessioni verso il mondo esterno: il **collettore**, la **base** e l'**emettitore**. Quando la tensione in ingresso scende sotto un valore critico, chiamato V_{in}, il transistor viene disabilitato e si comporta come una resistenza infinita[2]. La conseguenza è che l'output del circuito, V_{out}, assume un valore vicino a V_{cc}: una tensione regolata esternamente che, per questo tipo di transistor, vale generalmente +5 volt. Quando, al contrario, V_{in} supera il valore critico, il transistor si attiva e si comporta come un conduttore ideale[3], facendo scaricare V_{out} a terra (per convenzione, 0 volt).

È importante notare che quando V_{in} è basso, V_{out} è alto e viceversa. Questo circuito è quindi un invertitore che converte un valore logico 0 in un valore logico 1 e un valore logico 1 in un valore logico 0. La resistenza (la linea a zig-zag) è necessaria per limitare la quantità di corrente nel transistor ed evitare che esso si fonda. In genere il tempo richiesto per passare da uno stato a un altro è nell'ordine di pochi nanosecondi.

La Figura 3.1(b) mostra due transistor collegati in serie. Se V_1 e V_2 sono alte, allora entrambi i transistor saranno in conduzione e V_{out} sarà portato al valore basso, mentre, se entrambi gli ingressi sono bassi, allora i transistor corrispondenti saranno disattivati e l'uscita sarà alta. In altre parole V_{out} sarà alto se e soltanto se sia V_1 sia V_2 sono alte.

Nella Figura 3.1(c) i due transistor sono collegati in parallelo, invece che in serie. In questa configurazione se uno dei due ingressi è alto, allora il transistor corrispondente sarà attivato e l'uscita sarà scaricata a terra, mentre, se entrambi gli ingressi sono bassi, l'uscita rimarrà alta.

Questi tre circuiti, così come altri con lo stesso funzionamento, formano le tre porte logiche più semplici, chiamate rispettivamente NOT, NAND e NOR. Dato che spesso le porte NOT vengono chiamate **invertitori**, nel corso del testo utilizzeremo i due termini in modo equivalente. Se assumiamo la convenzione di interpretare "alto" (Vcc volt) come un valore logico 1 e "basso" (terra) come un valore logico 0, allora è possibile esprimere il valore in uscita come una funzione dei valori in ingresso. La Figura 3.2(a)-(c) mostra i simboli usati per indicare queste tre porte logiche e il loro comportamento funzionale. Nelle figure, A e B rappresentano gli ingressi, X l'uscita e ogni riga specifica il valore in uscita data una particolare combinazione dei valori in ingresso.

Se si fa passare il valore in uscita della Figura 3.1(b) in un circuito invertitore, si ottiene un nuovo circuito il cui comportamento è esattamente l'opposto di quello della porta

[2] Circuito aperto (*N.d.T.*).
[3] Corto circuito (*N.d.T.*).

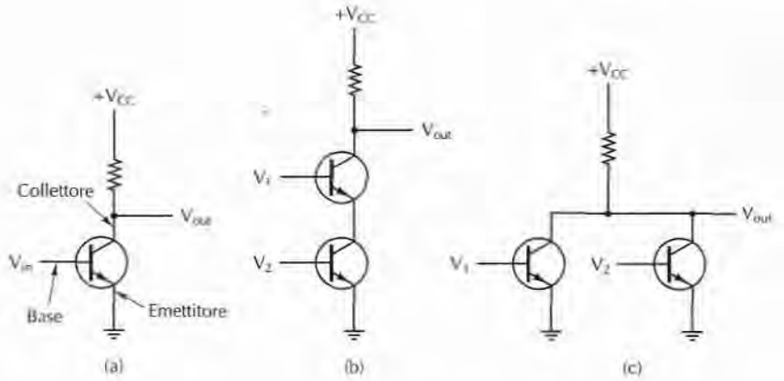


Figura 3.1 (a) Un invertitore. (b) Una porta NAND. (c) Una porta NOR.

logica NAND. In questo nuovo circuito l'uscita vale 1 se e solo se entrambi gli ingressi valgono 1; esso realizza la porta logica chiamata AND, il cui simbolo e la cui descrizione funzionale sono dati nella Figura 3.2(c). Analogamente è possibile collegare la porta logica NOR a un invertitore, in modo da ottenere un circuito la cui uscita valga 1 se almeno uno dei due ingressi vale 1, mentre valga 0 se entrambi gli ingressi valgono 0. La Figura 3.2(e) mostra il simbolo e la descrizione funzionale di questo circuito, chiamato porta logica OR. I piccoli cerchietti, utilizzati come parte dei simboli dell'invertitore, della porta NAND e della porta NOR, rappresentano sempre un'inversione.

Le porte logiche della Figura 3.2 sono i principali elementi costitutivi del livello logico digitale. Dall'analisi precedente è chiaro che le porte NAND e NOR necessitano di due transistor ciascuna, mentre le porte AND e OR ne richiedono tre; per questa ragione molti calcolatori sono basati sulle porte logiche NAND e NOR piuttosto che sulle più familiari porte AND e OR. In realtà tutte le porte logiche sono implementate in modo piuttosto diverso, ma in ogni caso le porte NAND e NOR rimangono più semplici di quelle AND e OR. Parlando di porte logiche vale la pena precisare che esse possono avere anche più di due ingressi; per fare un esempio una porta NAND potrebbe avere in teoria un numero di ingressi piuttosto elevato, anche se in pratica difficilmente ce ne sono più di otto.

Dato che ci si riferirà frequentemente alle principali tecnologie di costruzione delle porte logiche ne citiamo ora le principali, anche se questo argomento fa parte del livello dei dispositivi. Le due tecnologie principali sono quella **bipolare** e quella **MOS** (*Metal Oxide Semiconductor*, "semiconduttore metallo ossido"). Fra le bipolari, i tipi più importanti sono la **TTL** (*Transistor-Transistor Logic*), per anni il motore dell'elettronica digitale, e la **ECL** (*Emitter-Coupled Logic*), utilizzata quando è richiesto un funzionamento a velocità molto elevate. Attualmente nei circuiti dei calcolatori si impiega largamente la tecnologia MOS.

Le porte logiche MOS sono più lente delle TTL e delle ECL, ma richiedono molta meno potenza e hanno una dimensione decisamente inferiore, permettendo così di metterne insieme un numero elevato in uno spazio limitato. Esistono diverse varietà di porte MOS, tra cui le PMOS, le NMOS e le CMOS. I transistor MOS, pur essendo costruiti in modo differente rispetto ai transistor bipolari, sono tuttavia in grado di funzionare come degli in-

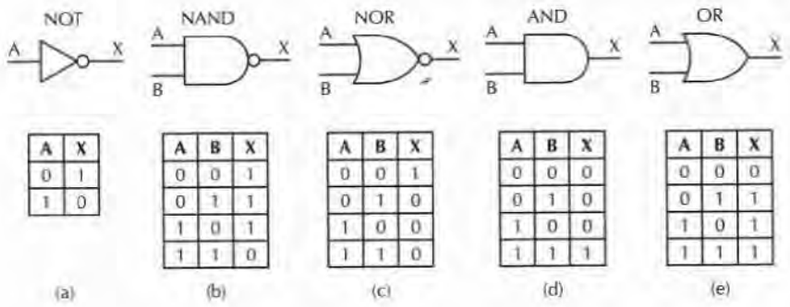


Figura 3.2 Simboli e comportamenti funzionali di cinque porte logiche elementari.

terruttori elettronici. La maggior parte delle CPU e delle memorie moderne utilizza la tecnologia CMOS che funziona a 3,3 volt. Questo è tutto ciò che occorre conoscere a proposito del livello dei dispositivi; i lettori interessati ad approfondire lo studio di questo livello possono consultare i riferimenti bibliografici forniti nel Capitolo 9.

3.1.2 Algebra di Boole

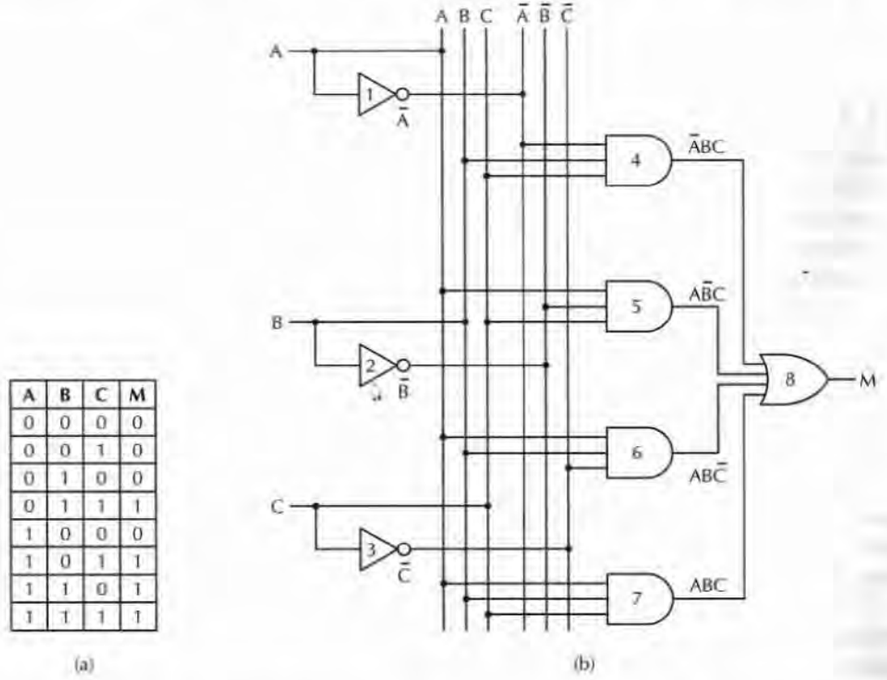
Per poter descrivere i circuiti costruiti combinando le porte logiche occorre definire un nuovo tipo di algebra in cui variabili e funzioni possono assumere soltanto i valori 0 e 1. Questa struttura viene chiamata **algebra di Boole**, in quanto è stata sviluppata dal matematico inglese George Boole (1815-1864). Per essere precisi, ci riferiamo in realtà a un particolare tipo di algebra booleana, chiamata **algebra booleana minimale**; ciononostante il termine "algebra booleana" è utilizzato in modo talmente esteso che non faremo alcuna distinzione.

Esattamente come esistono le funzioni nell'algebra ordinaria (cioè quella del liceo), esistono anche le funzioni nell'algebra booleana. Una funzione booleana ha una o più variabili di input e genera un valore di output che dipende soltanto dai valori di queste variabili. Una semplice funzione f può essere definita assumendo che $f(A)$ valga 1 se A vale 0 e $f(A)$ valga 0 se A vale 1; essa corrisponde alla funzione NOT della Figura 3.2(a).

Una funzione booleana di n variabili può essere completamente descritta mediante una tabella di 2^n righe; dato che esistono 2^n possibili combinazioni dei valori di input, ciascuna riga può quindi indicare il valore della funzione per una data combinazione degli input. Una tabella di questo tipo è chiamata **tabella di verità**; tutte le tabelle della Figura 3.2 ne sono un esempio. Se si decide di elencare sempre le righe di una tabella di verità in ordine numerico, cioè seguendo la sequenza 00, 01, 10 e 11 nel caso di due variabili, la funzione può essere completamente descritta dal numero binario a 2^n bit che si ottiene leggendo in verticale la colonna del risultato. Adottando questa convenzione, NAND è 1110, NOR è 1000, AND è 0001 e OR è 0111. Con due variabili esistono ovviamente soltanto 16 diverse funzioni booleane, corrispondenti alle 16 combinazioni possibili della stringa a 4 bit che rappresenta i risultati. Nell'algebra ordinaria esiste al contrario un numero infinito di funzioni a due variabili, nessuna delle quali può essere descritta dando una tabella che specifica i risultati per tutti i possibili valori di input. Nell'algebra ordinaria ogni variabile può infatti assumere un infinito numero di valori.

La Figura 3.3(a) mostra la tabella di verità di una funzione booleana a tre variabili: $M = f(A, B, C)$. Essa corrisponde alla funzione logica di maggioranza che vale 0 se la maggioranza dei suoi valori di input vale 0, mentre vale 1 se la maggioranza degli input vale 1. Anche se è possibile specificare in modo completo qualsiasi funzione booleana dandone la sua tabella di verità, all'aumentare del numero di variabili questa notazione diventa di dimensioni eccessivamente grandi. Per questo motivo si preferisce spesso adottare una notazione differente.

Per capire da dove nasce questa notazione alternativa si noti che ogni funzione booleana può essere descritta specificando quali combinazioni delle variabili di input producono, come risultato, 1. Nel caso della funzione mostrata nella Figura 3.3(a) ci sono quattro combinazioni di variabili di input che rendono M pari a 1. Per convenzione aggiungeremo una linea sopra una variabile di input per indicare che il suo valore è invertito (e non la apporremo in caso contrario). Utilizzeremo inoltre la notazione implicita della moltiplicazione, oppure un puntino, per denotare la funzione booleana AND (prodotto logico) e il simbolo + per indicare la funzione booleana OR (somma logica). Per esempio, seguendo queste convenzioni, $A\bar{B}C$ assume valore 1 solo quando A = 1, B = 0 e C = 1. Analogamente $\bar{A}B + B\bar{C}$ assume valore 1 solo quando (A = 1 e B = 0) oppure (B = 1 e C = 0). Le quattro righe della Figura 3.3(a) che producono in output il bit 1 sono: $\bar{A}BC$, $A\bar{B}C$, $AB\bar{C}$ e ABC; dato che la funzione M è vera



A	B	C	M
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Figura 3.3 (a) Tabella di verità della funzione di maggioranza di tre variabili. (b) Un circuito per (a).

(cioè vale 1) se una qualsiasi di queste quattro combinazioni è vera, è possibile riscrivere in modo più compatto la tabella di verità usando la seguente notazione

$$M = \bar{A}BC + A\bar{B}C + AB\bar{C} + ABC$$

Una funzione di n variabili può quindi essere descritta mediante una somma di un massimo di 2^n termini, ciascuno dei quali è costituito dal prodotto logico di n variabili.

Come vedremo tra poco questa formulazione è particolarmente importante in quanto porta direttamente a un'implementazione della funzione per mezzo di porte logiche di tipo standard.

È importante tenere a mente la differenza tra una funzione booleana astratta e la sua implementazione circuitale. Una funzione booleana consiste di variabili, come A, B e C, e operatori booleani come AND, OR e NOT, ed è descritta per mezzo di una tabella di verità o una funzione booleana, per esempio

$$F = A\bar{B}C + AB\bar{C}$$

Una funzione booleana può essere implementata mediante un circuito elettronico (che, in generale, può essere realizzato in vari modi), che utilizza segnali per rappresentare le variabili di input e di output oltre ad alcune porte logiche come AND, OR e NOT. In seguito, anche se talvolta potrà essere ambiguo, utilizzeremo la notazione AND, OR e NOT per riferirci agli operatori booleani e `AND`, `OR` e `NOT` per riferirci alle porte logiche.

3.1.3 Implementazione delle funzioni booleane

Come detto precedentemente è possibile implementare facilmente una funzione booleana se la si formula come una somma di prodotti. Utilizzando la Figura 3.3 come esempio possiamo vedere come si realizza questa implementazione. Nella Figura 3.3(b) gli input A, B e C sono rappresentati sul lato sinistro, mentre l'output della funzione è mostrato a destra. Dato che è necessario disporre dei complementi (inversi) delle variabili di input, essi sono generati intercettando gli input e facendoli passare attraverso gli invertitori indicati con i numeri 1, 2 e 3. Per rendere la figura più leggibile sono state disegnate sei linee verticali, tre delle quali connesse alle variabili di input e le restanti tre ai loro complementi. Queste linee forniscono un modo pratico per originare gli input delle porte logiche successive. Per esempio le porte 5, 6 e 7 usano A come input; in un circuito reale queste porte logiche potrebbero essere collegate direttamente ad A senza utilizzare alcun filo "verticale" intermedio.

Il circuito contiene quattro porte `AND`, una per ciascun termine che compone l'equazione di M (cioè una per ogni riga della tabella di verità in cui il bit della colonna risultato vale 1). Ciascuna porta `AND` calcola una riga della tabella di verità, com'è indicato nella figura. Il risultato finale viene poi calcolato effettuando l'`OR` fra tutti i prodotti.

Nel circuito mostrato nella Figura 3.3(b) è stata utilizzata una convenzione che verrà impiegata frequentemente nel corso del testo: l'incrocio fra due linee non implica alcuna connessione a meno che non sia presente un pallino nero nel punto di intersezione. Per esempio l'output della porta 3 incrocia tutte e sei le linee verticali, ma è connesso soltanto con $\bar{C}$. Dato che alcuni autori utilizzano convenzioni diverse, occorre prestare attenzione alla notazione impiegata.

Seguendo l'esempio della Figura 3.3 dovrebbe essere chiaro come si possa implementare un circuito che realizzi una qualsiasi funzione booleana.

1. Si scrive la tabella di verità della funzione.
2. Ci si munisce di invertitori per generare la negazione di ciascun input.
3. Si utilizza una porta `AND` per ciascun termine il cui valore nella colonna risultato vale 1.
4. Si collegano le porte `AND` agli input appropriati.
5. Si connettono tutti gli output delle porte `AND` nella porta `OR`.

Anche se abbiamo dimostrato la possibile implementazione di una qualsiasi funzione booleana mediante le porte `NOT`, `AND` e `OR`, spesso è più vantaggioso implementare circuiti utilizzando un solo tipo di porta logica. Fortunatamente esiste un modo diretto per convertire i circuiti generati dall'algoritmo precedente in uno che utilizzi unicamente la porta `NAND`, oppure la `NOR`. L'unica cosa che serve per effettuare una simile conversione è un modo per implementare le porte `NOT`, `AND` e `OR` utilizzando un solo tipo di porta logica. La Figura 3.4 mostra l'implementazione di queste tre porte utilizzando soltanto porte `NAND` (riga in alto), oppure soltanto porte `NOR` (riga in basso). Questo è il modo più diretto, ma esistono tuttavia anche altri metodi di conversione.

Un metodo per implementare una funzione booleana utilizzando soltanto le porte `NAND` o `NOR` consiste nel realizzarla inizialmente mediante le porte `NOT`, `AND` e `OR` seguendo

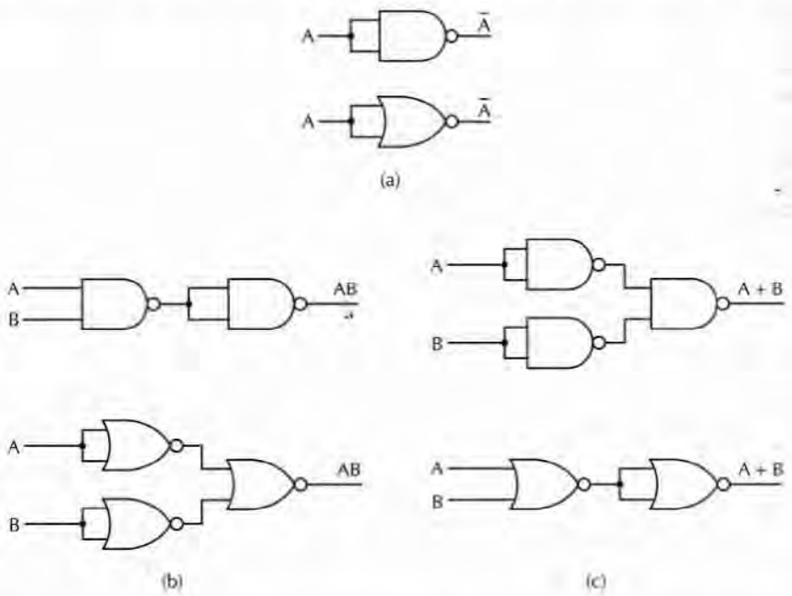


Figura 3.4 Costruzione delle porte logiche (a) `NOT`, (b) `AND` e (c) `OR` utilizzando soltanto porte `NOR`.

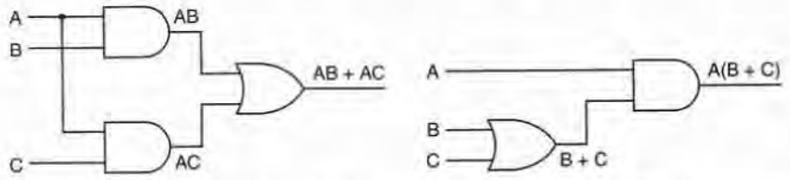
la precedente procedura. Successivamente occorre sostituire le porte logiche con più ingressi con circuiti equivalenti composti da porte con due soli ingressi. $A + B + C + D$ può per esempio essere calcolata come $(A + B) + (C + D)$ utilizzando tre porte logiche a due ingressi. Infine le porte NOT, AND e OR devono essere sostituite dai circuiti della Figura 3.4.

Questa procedura mostra che esiste sempre una possibile implementazione; tuttavia i circuiti ottenuti non sono ottimali, nel senso che non utilizzano il minimo numero possibile di porte logiche. Sia le porte NAND sia le porte NOR costituiscono un insieme di connettivi **funzionalmente completo**, nel senso che una qualsiasi funzione booleana può essere calcolata usando soltanto uno di questi due tipi di porte. Nessun'altra porta logica gode di questa proprietà; questo è un ulteriore motivo per il quale NAND e NOR sono spesso utilizzate come elementi base nella costruzione dei circuiti.

3.1.4 Equivalenza di circuiti

Spesso i progettisti cercano di ridurre il numero di porte logiche utilizzate nei loro prodotti per limitare il costo dei componenti, le dimensioni delle schede con i circuiti stampati, il consumo energetico e altri fattori. Per ridurre la complessità di un circuito un progettista deve trovare un altro circuito che calcoli la stessa funzione di quello originario, ma che sia composto da un numero inferiore di porte (oppure da porte più semplici, per esempio porte a due ingressi al posto di porte a quattro ingressi).

Come esempio di utilizzo di tecniche dell'algebra booleana per la ricerca di circuiti equivalenti, consideriamo il circuito e la tabella di verità per la funzione $AB + AC$ mostrati nella Figura 3.5(a). Molte delle regole dell'algebra ordinaria valgono anche per l'algebra booleana;



A	B	C	AB	AC	AB + AC
0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	0	0	0
0	1	1	0	0	0
1	0	0	0	0	0
1	0	1	0	1	1
1	1	0	1	0	1
1	1	1	1	1	1

(a)

A	B	C	A	B + C	A(B + C)
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	1	0
0	1	1	0	1	0
1	0	0	1	0	0
1	0	1	1	1	1
1	1	0	1	1	1
1	1	1	1	1	1

(b)

Figura 3.5 Due funzioni equivalenti. (a) $AB + AC$. (b) $A(B + C)$.

in particolare, usando la proprietà distributiva, è possibile fattorizzare $AB + AC$ nella forma $A(B + C)$, la cui tabella di verità e il cui circuito sono mostrati nella Figura 3.5(b). Dato che due funzioni sono equivalenti se e solo se hanno lo stesso valore di output per tutti i possibili input, è facile verificare dalle tabelle di verità della Figura 3.5 che $A(B + C)$ e $AB + AC$ sono effettivamente equivalenti. Pur essendo equivalenti il circuito della Figura 3.5(b) è chiaramente migliore di quello della Figura 3.5(a), in quanto è composto da un numero minore di porte logiche.

Di solito un progettista di circuiti parte da una formula e in seguito, applicando le leggi dell'algebra di Boole, cerca di semplificarla. Dalla funzione finale può infine essere costruito il circuito.

Per utilizzare questo approccio occorre conoscere alcune identità: le principali sono elencate nella Figura 3.6. È interessante notare che ciascuna legge compare in due forme, **duali** l'una rispetto all'altra. Scambiando AND con OR e anche 1 con 0 è possibile creare una delle due forme a partire dall'altra. È possibile verificare tutte le leggi costruendo le loro tabelle di verità; i risultati sono ragionevolmente intuitivi per tutte le leggi tranne che per la legge di De Morgan, la proprietà di assorbimento e la legge distributiva della somma rispetto al prodotto. La legge di De Morgan può inoltre essere estesa a più di due variabili, per esempio $\overline{ABC} = \overline{A} + \overline{B} + \overline{C}$.

La legge di De Morgan suggerisce una notazione alternativa. La Figura 3.7(a) mostra la forma AND in cui la negazione è indicata, sia per gli input sia per gli output, mediante i cerchietti di inversione. Una porta logica OR con gli input invertiti è quindi equivalente a una porta NAND. Dalla Figura 3.7(b), che rappresenta la forma duale della legge di De Morgan, dovrebbe essere chiaro che una porta NOR può essere disegnata come una porta AND con gli input invertiti. Negando entrambe le forme della legge di De Morgan si ottengono le Figure 3.7(c) e (d), che mostrano due rappresentazioni equivalenti delle porte logiche AND e OR. Esistono simboli analoghi per le forme a più variabili della legge di De Morgan (per esempio, una porta NAND a n input diventa una porta OR con n input invertiti).

La conversione della tabella di verità dalla rappresentazione somma-di-prodotti alla forma pura NAND o NOR risulta agevolata dalle identità della Figura 3.7 e dalle corrispondenti versioni per le porte a più ingressi. Come esempio consideriamo la funzione XOR

Nome	Forma AND	Forma OR
Elemento neutro	$1A = A$	$0 + A = A$
Assorbimento	$0A = 0$	$1 + A = 1$
Idempotenza	$AA = A$	$A + A = A$
Complementazione	$A\overline{A} = 0$	$A + \overline{A} = 1$
Proprietà commutativa	$AB = BA$	$A + B = B + A$
Proprietà associativa	$(AB)C = A(BC)$	$(A + B) + C = A + (B + C)$
Proprietà distributiva	$A + BC = (A + B)(A + C)$	$A(B + C) = AB + AC$
Legge di assorbimento	$A(A + B) = A$	$A + AB = A$
Legge di De Morgan	$\overline{AB} = \overline{A} + \overline{B}$	$\overline{A + B} = \overline{A}\,\overline{B}$

Figura 3.6 Identità dell'algebra booleana.

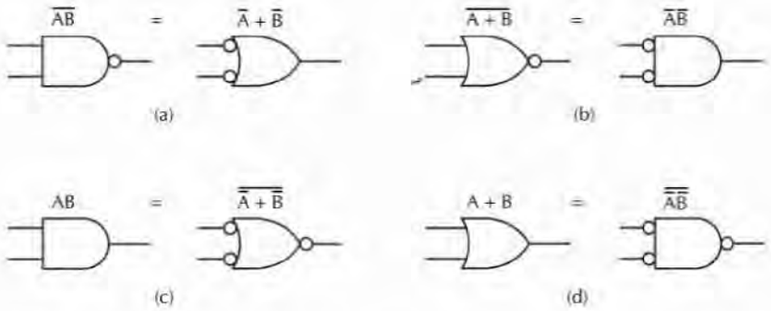
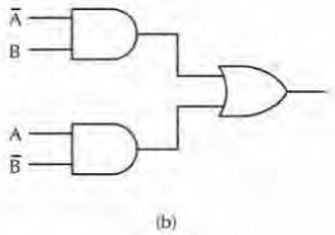


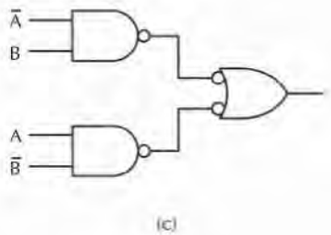
Figura 3.7 Simboli equivalenti per alcune porte logiche: (a) NAND. (b) NOR. (c) AND. (d) OR.

(OR esclusivo) della Figura 3.8(a), il cui circuito nella forma standard somma-di-prodotti è rappresentato nella Figura 3.8(b). Per convertirlo nella forma NAND bisognerebbe ridisegnare le linee che collegano l'output delle porte AND all'input della porta OR utilizzando due cerchietti d'inversione, come illustra la Figura 3.8(c); infine, utilizzando l'equivalenza mostrata nella Figura 3.7(a) si ottiene il circuito della Figura 3.8(d). Le variabili $\bar{A}$ e $\bar{B}$ possono essere generate da A e B utilizzando le porte NAND o NOR con i loro input legati insieme. Si noti che i cerchietti d'inversione possono essere spostati a piacere lungo una linea, muovendoli per esempio dagli output delle porte di input della Figura 3.8(d) agli input della porta di output.

A	B	XOR
0	0	0
0	1	1
1	0	1
1	1	0

(a)





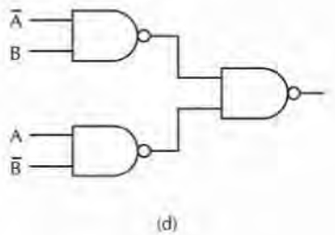


Figura 3.8 (a) Tabella di verità della funzione XOR. (b)-(d) Tre circuiti per calcolarla.

A	B	F
0	0	0
0	5	0
5	0	0
5	5	5

(a)

A	B	F
0	0	0
0	1	0
1	0	0
1	1	1

(b)

A	B	F
1	1	1
1	0	1
0	1	1
0	0	0

(c)

Figura 3.9 (a) Caratteristiche elettriche di un dispositivo. (b) Logica positiva. (c) Logica negativa.

Come ultima osservazione sull'equivalenza di circuiti dimostreremo che, in modo sorprendente, una stessa porta logica reale può calcolare diverse funzioni a seconda delle convenzioni adottate. La Figura 3.9(a) mostra l'output di una certa porta, F, al variare della combinazione degli input (input e output sono indicati in volt). Se adottiamo la convenzione, chiamata **logica positiva**, secondo la quale 0 volt è il valore logico 0 e 3,3 volt (o 5 volt) è il valore logico 1, otteniamo la tabella di verità della Figura 3.9(b), corrispondente alla funzione AND. Se al contrario adottiamo la **logica negativa**, per la quale 0 volt rappresenta il valore logico 1 e 3,3 volt (o 5 volt) il valore logico 0, otteniamo la tabella di verità mostrata nella Figura 3.9(c), corrispondente alla funzione OR.

La convenzione scelta per assegnare le tensioni ai valori logici riveste quindi un'importanza cruciale. Se non specificato diversamente, d'ora in poi utilizzeremo la logica positiva; i termini valore logico 1, vero e alto saranno quindi fra loro sinonimi, così come lo saranno i termini valore logico 0, falso e basso.

3.2 Circuiti logici digitali elementari

Nei paragrafi precedenti abbiamo visto come implementare tabelle di verità e altri semplici circuiti utilizzando singole porte logiche. Anche se una volta era comune farlo, attualmente solo pochi circuiti sono realmente costruiti "porta per porta". Al giorno d'oggi nella costruzione di circuiti i componenti elementari sono generalmente rappresentati da moduli contenenti un certo numero di porte. Nei paragrafi successivi analizzeremo più da vicino questi blocchi elementari e vedremo come vengono utilizzati e come sono costruiti a partire da singole porte logiche.

3.2.1 Circuiti integrati

Le porte logiche non sono prodotte o vendute individualmente, ma in unità chiamate **circuiti integrati**, alle quali spesso ci si riferisce con i termini **IC** o **chip**. Un IC è un quadrato di silicio di circa 5 mm × 5 mm sul quale sono state posizionate alcune porte. Di solito i piccoli IC sono montati in un contenitore rettangolare di plastica o ceramica largo dai 5 ai 15 mm e lungo dai 20 ai 50 mm. Sul lato più lungo vi sono due file parallele di contatti (*pin*) lunghi circa 5 mm, inseribili in una presa (*socket*) oppure saldabili su schede di circuiti stampati. Ciascun pin è collegato a un input o a un output di una porta logica oppure al-

l'alimentazione o alla terra. Con il termine tecnico **DIP** (*Dual Inline Package*, "contenitore a due file di piedini") ci si riferisce al contenitore che all'esterno presenta due file di pin e al suo interno alcuni IC; tutti però lo chiamano chip, rendendo in questo modo più sfocata la distinzione tra il pezzo di silicio e il suo contenitore. I contenitori più comuni hanno 14, 16, 18, 20, 22, 24, 28, 40, 64 o 68 pin. Spesso nel caso di chip di grandi dimensioni si utilizzano contenitori quadrati con i pin disposti su tutti e quattro i lati o sulla parte inferiore.

Come indicato nella lista sottostante, i chip possono essere approssimativamente classificati in base al numero di porte che contengono. Anche se questa classificazione è decisamente grossolana, talvolta si rivela utile.

Circuiti SSI (*Small Scale Integrated*): da 1 a 10 porte.
Circuiti MSI (*Medium Scale Integrated*): da 10 a 100 porte.
Circuiti LSI (*Large Scale Integrated*): da 100 a 100.000 porte.
Circuiti VLSI (*Very Large Scale Integrated*): più di 100.000 porte.

Queste classi hanno proprietà diverse e vengono utilizzate in modi differenti.

Un chip SSI contiene in genere tra le due e le sei porte logiche, ciascuna delle quali può essere utilizzata individualmente seguendo le modalità presentate nei paragrafi precedenti. La Figura 3.10 rappresenta lo schema di un comune chip SSI con quattro porte NAND. Dato che ciascuna di queste porte ha due input e un output sono necessari 12 pin per le quattro porte. Inoltre il chip necessita della tensione (V_{cc}) e della terra (GND), che sono condivise da tutte le porte. Di solito il contenitore ha una tacca vicino al pin 1 per poterne riconoscere l'orientamento. Per convenzione, al fine di rendere più leggibili i diagrammi dei circuiti, non si mostrano l'alimentazione, la terra e neppure le porte inutilizzate.

Al costo di pochi centesimi l'uno sono disponibili molti altri chip simili al precedente. Ciascun chip SSI contiene una manciata di porte e fino a una ventina di pin. Negli anni '70 i calcolatori erano costruiti con un gran numero di chip di questo tipo, mentre al giorno d'oggi sono stampate all'interno di un unico chip un'intera CPU e una significativa quantità di memoria (cache).

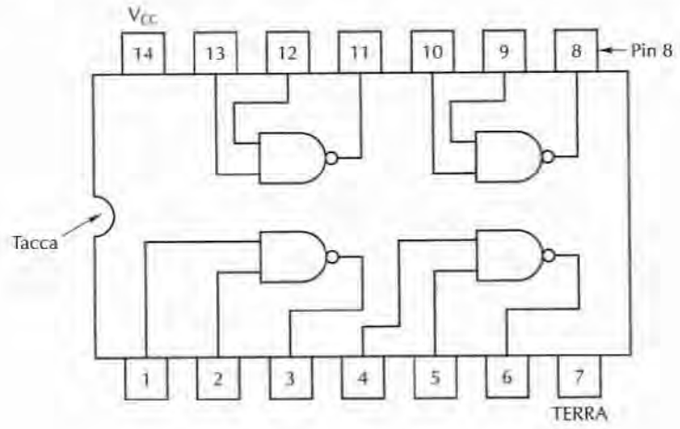


Figura 3.10 Chip SSI con quattro porte logiche.

Per i nostri scopi tutte le porte logiche considerate sono ideali, nel senso che l'output appare non appena viene applicato il voltaggio in input. In realtà i chip hanno un ritardo temporale finito, chiamato **ritardo della porta**, che comprende sia il tempo dovuto alla propagazione del segnale attraverso il chip, sia il tempo di commutazione. Generalmente questi ritardi sono compresi tra 1 e 10 ns.

Lo stato dell'arte attuale della tecnologia permette di inserire fino a 10 milioni di transistor all'interno di un chip. Dato che ogni circuito può essere costruito mediante porte NAND, si potrebbe pensare che un produttore potrebbe realizzare un chip estremamente generalizzato contenente 5 milioni di porte NAND. Sfortunatamente però un chip di questo tipo richiederebbe 15.000.000 pin; considerando che la distanza standard tra i pin è di 0,1 pollici, il chip supererebbe i 19 km, il che potrebbe porre dei limiti alla sua commercializzazione. È chiaro che l'unico modo per trarre vantaggio dallo stato attuale della tecnologia consiste nel progettare circuiti con un elevato rapporto porte/pin. Nei paragrafi successivi esamineremo alcuni semplici MSI che, per implementare una determinata funzione, combinano un buon numero di porte interne pur richiedendo un limitato numero di connessioni esterne.

3.2.2 Reti combinatorie

Molte applicazioni della logica digitale richiedono un circuito, chiamato **rete combinatoria**, con più input e più output, in cui gli output sono unicamente determinati dagli input. Non tutti i circuiti hanno questa proprietà; un circuito contenente elementi di memoria può per esempio generare valori che dipendono non solo dalle variabili d'ingresso, ma anche dai valori memorizzati. Un circuito che implementa una tabella di verità, come quella della Figura 3.3(a), è un tipico esempio di rete combinatoria. In questo paragrafo esamineremo le reti combinatorie utilizzate più di frequente.

Multiplexer

In logica digitale un ***multiplexer*** è un circuito con 2^n dati di input, un valore di output e n input di controllo; gli input di controllo permettono di selezionare uno dei dati di input, che viene "instradato" verso l'output. La Figura 3.11 mostra un diagramma di un multiplexer con otto input. Le tre linee di controllo, A, B e C, codificano un numero a 3 bit che specifica quale delle otto linee di input deve essere instradata verso la porta OR e quindi verso l'output. Indipendentemente dal valore definito dalle linee di controllo, sette delle porte AND genereranno sempre il valore 0, mentre quella rimanente produrrà in output 0 oppure 1, a seconda del valore della linea d'ingresso selezionata. Ciascuna porta AND può essere abilitata da una diversa combinazione degli input di controllo. Il circuito del multiplexer è mostrato nella Figura 3.11; aggiungendo anche la tensione e la terra può essere confezionato in un contenitore dotato di 14 pin.

La Figura 3.12(b) mostra come possiamo utilizzare il multiplexer per implementare la funzione di maggioranza della Figura 3.3(a). In corrispondenza di ciascuna combinazione di A, B e C viene selezionata una delle linee dei dati di input e ciascuna di loro è collegata a V_{cc} (valore logico 1) oppure alla terra (valore logico 0). L'algoritmo che permette di collegare gli input in modo corretto è semplice: l'input D_i è lo stesso valore presente nella riga i della tabella di verità. Nella Figura 3.3(a) le righe 0, 1, 2 e 4 valgono 0 e quindi gli input corrispondenti sono collegati alla terra; le righe rimanenti valgono invece 1 e sono quindi

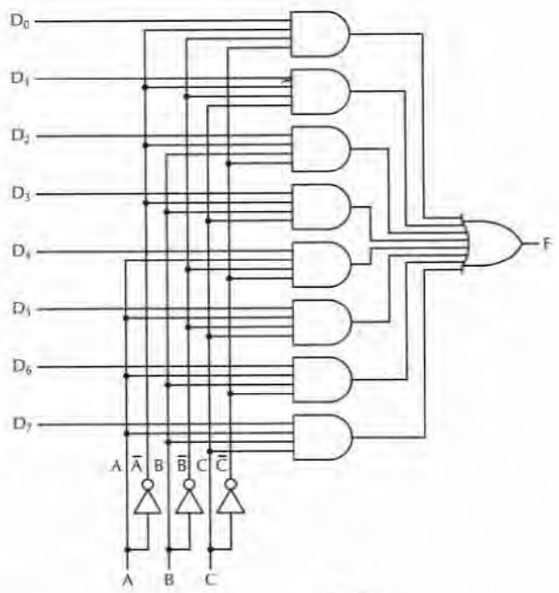


Figura 3.11 Multiplexer a otto vie.

collegate al valore logico 1. Seguendo questa strategia è possibile implementare una qualsiasi tabella di verità mediante il chip della Figura 3.12(a).

Finora abbiamo visto come utilizzare un multiplexer per selezionare un input fra alcuni valori d'ingresso e per implementare una tabella di verità. Un'altra delle possibili applicazioni

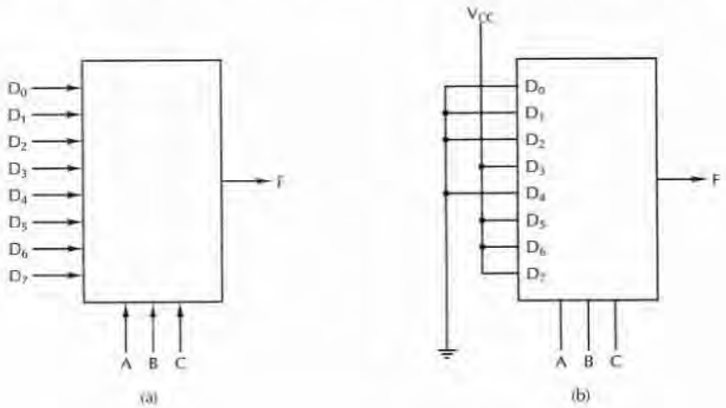


Figura 3.12 (a) Multiplexer MSI. (b) Lo stesso multiplexer per calcolare la funzione di maggioranza.

è la conversione di dati da parallelo a seriale. Se si immettono 8 bit di dati nelle linee di input e se si fa variare il valore (binario) delle linee di controllo da 000 a 111, si ottengono sulla linea di output gli 8 bit in serie. Un utilizzo molto comune della conversione da parallelo a seriale lo si riscontra nelle tastiere: la pressione di ogni tasto definisce implicitamente un numero a 7 o 8 bit che deve essere spedito serialmente lungo una linea telefonica.

L'inverso del multiplexer è il **demultiplexer**, che redirige il suo segnale di input verso uno dei 2^n output in base ai valori delle linee di controllo; se il valore binario definito dalle linee di controllo è k, viene selezionato l'output k.

Decodificatori

Come secondo esempio analizzeremo ora un circuito, chiamato **decodificatore** (*decoder*), che accetta come input un numero a n bit e lo utilizza per impostare a 1 una sola delle 2^n linee di output. La Figura 3.13 mostra questo circuito nel caso $n = 3$.

Per capire in quali situazioni può essere utile questo circuito immaginiamo una piccola memoria di otto chip, da 1 MB ciascuno. Gli indirizzi del chip 0 variano da 0 a 1 MB, quelli del chip 1 da 1 MB a 2 MB, e così via. Quando si fornisce alla memoria un indirizzo, si utilizzano i suoi 3 bit più significativi per selezionare uno degli otto chip. In riferimento al circuito della Figura 3.13 questi 3 bit corrispondono ai tre input, A, B e C; a seconda del loro valore solo una delle linee di output, assume il valore 1, mentre le altre rimangono a 0. Ciascuna linea di output permette poi di abilitare uno degli otto chip della memoria; dato che solo una linea di output viene impostata al valore 1, solo uno dei chip viene abilitato.

Il funzionamento del circuito della Figura 3.13 è semplice. Ciascuna porta AND ha tre input, il primo dei quali è A o $\overline{A}$, il secondo B o $\overline{B}$ e il terzo C o $\overline{C}$, e viene abilitata da una diversa combinazione dei valori di input: D_0 da $\overline{A}\ \overline{B}\ \overline{C}$, D_1 da $\overline{A}\ \overline{B}\ C$, e così via.

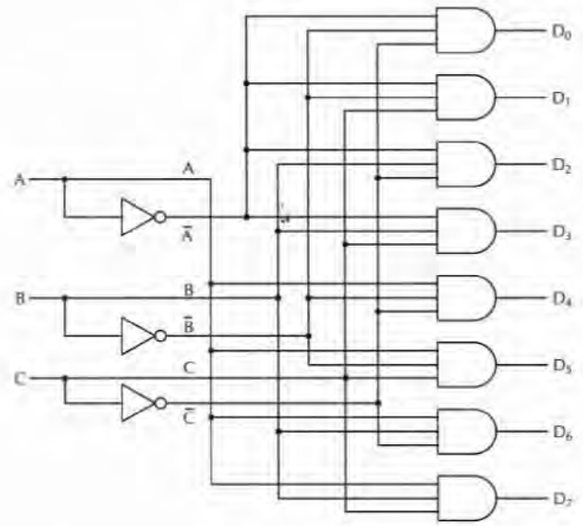


Figura 3.13 Decodificatore da 3 a 8.

Comparatori

Un altro circuito particolarmente utile è il **comparatore**, che permette di confrontare due stringhe di bit. Il semplice comparatore mostrato nella Figura 3.14 accetta due input, *A* e *B*, ciascuno lungo 4 bit, e genera 1 se sono uguali, mentre 0 se sono diversi. Il circuito è basato sulla porta logica XOR (EXCLUSIVE OR), che produce in output un valore 0 se i suoi input sono uguali e un valore 1 se sono diversi. Se due stringhe in ingresso sono uguali, tutte e quattro le porte XOR devono generare come risultato 0. Questi quattro segnali possono poi essere connessi a una stessa porta logica OR in modo da produrre un valore 0 quando gli input sono uguali e un valore 1 nel caso contrario. Nel nostro esempio abbiamo utilizzato una porta NOR nell'ultimo stadio del circuito in modo da invertire il risultato del test: 1 significa uguale, mentre 0 significa diverso.

Array logici programmabili

Un chip molto generale che permette di calcolare somme di prodotti è l'**array logico programmabile** o **PLA** (*Programmable Logic Array*), di cui la Figura 3.15 mostra un semplice esempio. Questo chip ha 12 ingressi e al suo interno questi valori vengono invertiti; quindi il numero totale di segnali di input diventa 24. Il cuore del circuito è costituito da una schiera di 50 porte AND che possono avere come input un qualsiasi sottoinsieme dei 24 segnali di input. Una matrice di 24 × 50 bit fornita dall'utente determina le connessioni desiderate tra i segnali di input e le 50 porte AND. Ogni linea di input connessa alle 50 porte AND contiene un fusibile. Al momento della fabbricazione del chip i 1200 fusibili sono intatti; successivamente l'utente può programmare la matrice bruciando i fusibili con l'applicazione di un'alta tensione.

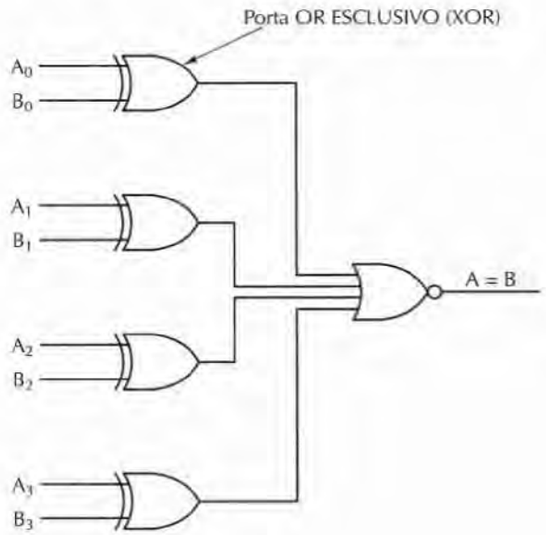


Figura 3.14 Semplice comparatore a 4 bit.

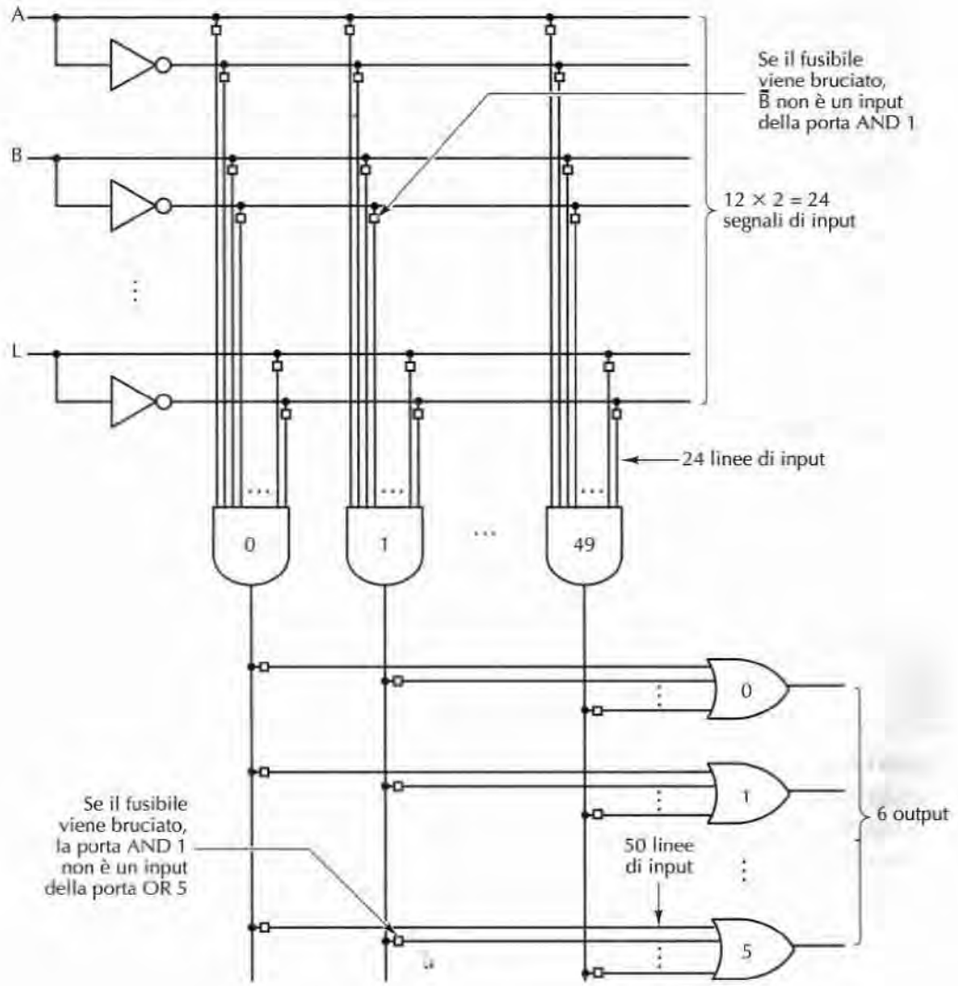


Figura 3.15 Array logico programmabile a 12 input e 6 output. I quadratini rappresentano i fusibili che possono essere bruciati per determinare la funzione da calcolare. I fusibili sono disposti in due matrici: quella superiore per le porte AND, quella inferiore per le porte OR.

L'uscita del circuito consiste in 6 porte OR, che possono avere fino a 50 input, corrispondenti agli output delle porte AND. Anche in questo caso l'utente deve fornire una matrice (50 × 6) per specificare quali connessioni instaurare. Il chip ha 20 pin in tutto, 12 per gli input, 6 per gli output, e 2 per la tensione e la terra.

Come esempio di utilizzo di un circuito PLA consideriamo nuovamente il circuito mostrato nella Figura 3.3(b), che ha tre input, quattro porte AND, una porta OR e tre invertitori.

Il nostro PLA, dopo aver impostando inizialmente le connessioni in modo appropriato, può calcolare la stessa funzione usando tre dei suoi 12 input, quattro delle 50 porte AND e una delle sue 6 porte OR. Le quattro porte AND dovrebbero calcolare rispettivamente $\overline{A}BC$, $A\overline{B}C$, $AB\overline{C}$ e ABC, e la porta OR dovrebbe accettare in input questi quattro prodotti. In realtà lo stesso PLA potrebbe essere utilizzato per calcolare quattro funzioni di analoga complessità. Nel caso di funzioni semplici il numero di variabili rappresenta un fattore vincolante, mentre per funzioni più complesse il limite potrebbe essere rappresentato dal numero di porte AND o OR.

Anche se i PLA *programmabili sul campo*, come quello appena descritto, vengono ancora utilizzati, oggi, in molte applicazioni, si preferisce impiegare dei PLA costruiti ad hoc. Questi ultimi sono più economici di quelli programmabili e, in caso di grandi volumi, vengono progettati sulla base delle specifiche del cliente.

Alla luce della presentazione dei tre modi diversi per implementare la tabella di verità della Figura 3.3(a) possiamo passare a un confronto. Utilizzando componenti SSI sono necessari 4 chip; una soluzione alternativa, mostrata nella Figura 3.12(b), prevede invece di utilizzare un unico multiplexer di tipo MSI. Infine l'ultima possibilità consiste nell'usare un quarto di un chip PLA. Se occorre implementare più funzioni il chip PLA è ovviamente più efficiente rispetto agli altri due metodi; nel caso invece di circuiti semplici, i chip SSI e MSI sono preferibili in quanto più economici.

3.2.3 Circuiti per l'aritmetica

È arrivato ora il momento di passare dai circuiti MSI di uso generale alle reti combinatorie impiegate per eseguire calcoli aritmetici. Inizieremo con un semplice registro a scorrimento (*shifter*) a 8 bit, continueremo guardando com'è costruito un sommatore e infine esamineremo le unità aritmetico-logiche che svolgono un ruolo centrale all'interno di tutti i calcolatori.

Registri a scorrimento

Il primo circuito aritmetico MSI che analizziamo ha otto input e otto output (vedi la Figura 3.16). Gli input sono collegati alle linee D_0, ..., D_7, mentre l'output, corrispondente all'input traslato di un bit, risulta disponibile sulle linee S_0, ..., S_7. La linea di controllo, C,

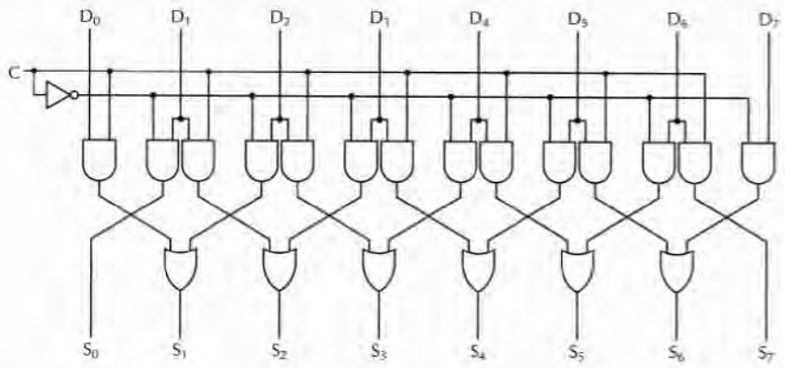


Figura 3.16 Registro a scorrimento a sinistra di 1 bit.

ha valore 0 se lo spostamento deve avvenire verso sinistra, e 1 in caso contrario. Nel caso di uno spostamento a sinistra si inserisce uno 0 nel bit 7, e nel caso di uno shift a destra si inserisce un 1 nel bit 0.

Per comprendere il funzionamento del circuito si noti che per tutti i bit c'è una coppia di porte AND, fatta eccezione per le porte che si trovano alle estremità. Quando $C = 1$ la porta che si trova a destra in ciascuna coppia viene abilitata, lasciando passare il bit corrispondente verso l'output. Dato che la porta AND è collegata all'input della porta OR alla sua destra, si ottiene uno scorrimento verso destra. Quando $C = 0$ è la porta AND di sinistra in ciascuna coppia a essere abilitata, producendo uno spostamento verso sinistra.

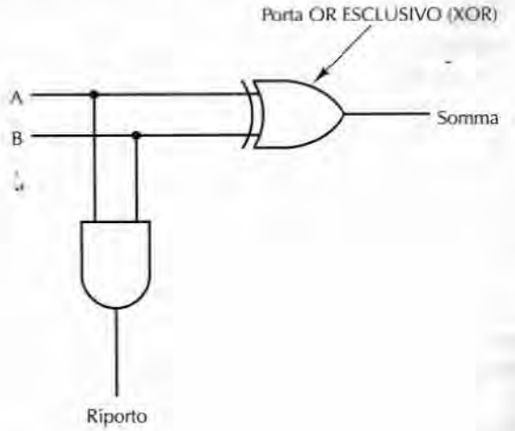
Sommatori

È quasi impossibile immaginare un calcolatore che non sia in grado di eseguire le somme. Per questo motivo un circuito che effettui l'addizione è una parte di essenziale importanza per qualsiasi CPU. La Figura 3.17(a) mostra la tabella di verità per la somma di interi a 1 bit, in cui sono presenti due output: la somma dei due input, A e B, e il riporto da sommare alla successiva posizione (a sinistra). La Figura 3.17(b) rappresenta un circuito, chiamato **half adder** (cioè *semisommatore*, non tenendo conto del riporto in ingresso), capace di calcolare il bit della somma e il bit del riporto.

Questo circuito è in grado di sommare correttamente i bit meno significativi di due stringhe binarie, ma non è in grado di eseguire correttamente la somma degli altri bit, dato che non riesce a gestire il riporto che arriva dalle posizioni precedenti. Per far ciò è necessario un **sommatore** come quello mostrato nella Figura 3.18. Osservando attentamente il circuito dovrebbe apparire chiaro che un sommatore è costruito a partire da due semisommatori. La linea di output *Sum* vale 1 se una o tre delle linee A, B e *Carry in* valgono 1. *Carry out* vale 1 se sia A sia B valgono 1 (input di sinistra della porta OR) oppure se uno dei due

A	B	Somma	Riporto
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

(a)

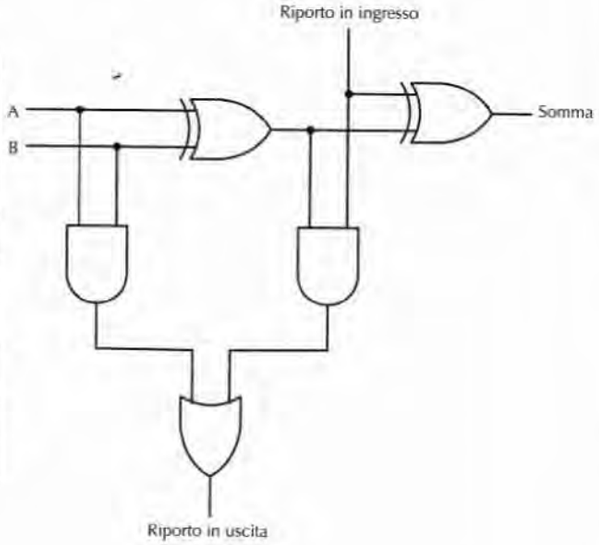


(b)

Figura 3.17 (a) Tabella di verità della somma di 1 bit. (b) Circuito di un semi-sommatore.

A	B	Riporto in ingresso	Somma	Riporto in uscita
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

(a)



(b)

Figura 3.18 (a) Tabella di verità di un sommatore. (b) Circuito di un sommatore.

vale 1 e anche *Carry in* vale 1. L'unione dei due semisommatori permette di generare in output sia il bit della somma sia il bit del riporto.

Per esempio, per generare un sommatore di due parole a 16 bit occorre replicare il circuito della Figura 3.18(b) per 16 volte. Per un dato bit il riporto in uscita viene utilizzato come riporto in entrata per il bit alla sua sinistra. Il riporto in entrata del bit all'estremità destra è collegato al valore 0. Questo tipo di sommatore è chiamato **sommatore a propagazione di riporto**, dato che nel caso peggiore, sommando 1 a 111...111 (binario), la somma non può essere completata finché il riporto non si sia propagato lungo tutta la parola, dal bit all'estremità destra fino a quello all'estremità sinistra. Esistono anche altri tipi di sommatori che non hanno questo ritardo e risultano dunque più veloci; per questo motivo vengono generalmente preferiti rispetto a quelli a propagazione di riporto.

Per fare un esempio di sommatore più veloce immaginiamo di dividere un sommatore a 32 bit in uno per i 16 bit meno significativi e in un altro per i 16 bit più significativi. Nel momento in cui comincia l'addizione quest'ultimo non può ancora iniziare a lavorare poiché è obbligato ad attendere il valore del riporto per un tempo pari a 16 somme di singoli bit.

Consideriamo però la seguente modifica. Invece di avere un singolo sommatore per i bit più significativi ne creiamo due identici in parallelo duplicando l'hardware corrispondente. Il circuito così modificato consiste in tre sommatori a 16 bit: uno per i bit meno significativi e due per quelli più significativi, *U0* e *U1*, funzionanti in parallelo. Nel sommatore *U0* viene collegato il valore 0 come riporto, mentre nel sommatore *U1* viene collegato il valore 1. Tutti e tre i sommatori possono ora cominciare nello stesso momento, anche se

soltanto uno dei due sommatori *U0* e *U1* sarà corretto. Dopo aver atteso il completamento delle prime 16 somme di singoli bit si conoscerà il valore del riporto da aggiungere al sommatore dei 16 bit più significativi. Grazie a questo valore è possibile selezionare il sommatore corretto fra *U0* e *U1*. Questo trucco permette di dimezzare il tempo richiesto dall'addizione; un simile sommatore viene chiamato a **selezione del riporto**. Lo stesso trucco può essere ripetuto per costruire i sommatori a 16 bit come più sommatori a 8 bit, e così via.

Unità aritmetico logiche

La maggior parte dei calcolatori contiene un unico circuito in grado di effettuare le operazioni AND, OR e somma di due parole. Di solito un circuito di questo tipo funzionante su parole a n bit è costruito a partire da n identici circuiti per le singole posizioni dei bit. La Figura 3.19 è un esempio di tale circuito, chiamato **unità aritmetico logica** o **ALU** (*Arithmetic Logic Unit*); esso può calcolare una qualsiasi delle quattro funzioni, A AND B, A OR B, $\overline{B}$ oppure $A + B$, in base al valore binario (00, 01, 10 oppure 11) definito dalle linee F_0 e F_1 preposte alla selezione della funzione aritmetica. Si noti che con $A + B$ si intende la somma aritmetica di A e B e non l'operazione booleana OR.

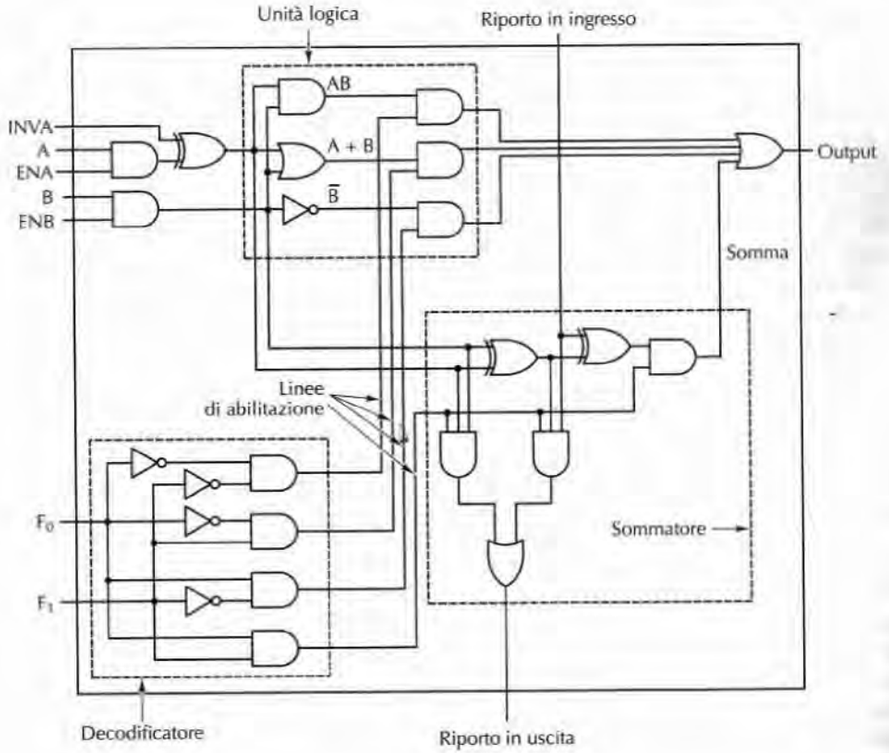


Figura 3.19 ALU a 1 bit.

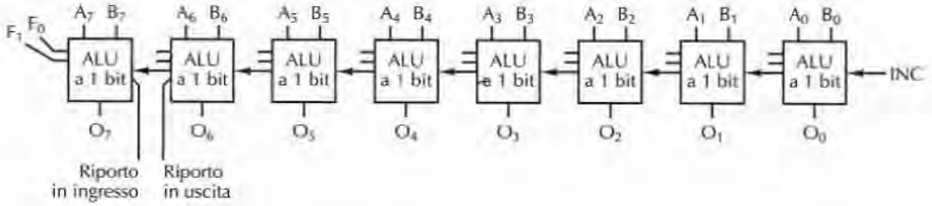


Figura 3.20 Otto ALU a 1 bit connesse per comporre una ALU a 8 bit. Per semplificare non sono mostrati segnali di abilitazione e segnali di inversione.

Il settore in basso a sinistra della nostra ALU contiene un decodificatore a 2 bit che permette di generare, in base ai segnali di controllo F_0 ed F_1, i segnali di attivazione delle quattro operazioni. In base ai valori di F_0 e F_1 viene selezionata una sola delle quattro linee di attivazione, permettendo all'output della funzione selezionata di passare attraverso la porta logica OR che genera l'output finale.

Il settore in alto a sinistra contiene le porte logiche necessarie per calcolare A AND B, A OR B e $\overline{B}$; in base alle linee di attivazione che escono dal decodificatore uno solo di questi risultati viene passato alla porta logica finale OR. Dato che solo uno degli output del decodificatore varrà 1, soltanto una delle quattro porte AND che forniscono i valori d'ingresso alla porta OR verrà attivata; l'output delle altre sarà invece 0, indipendentemente dai valori di A e B. Oltre a poter usare A e B come input per le operazioni logiche o aritmetiche, è anche possibile forzare uno dei due valori a 0 negando rispettivamente ENA oppure ENB. È inoltre possibile ottenere il valore di $\overline{A}$ abilitando il segnale INVA. Nel Capitolo 4 vedremo alcuni utilizzi di INVA, ENA e ENB. In condizioni normali entrambi gli input sono abilitati; ENA e ENB valgono quindi 1, mentre INVA è impostato a 0. In questa situazione A e B sono immessi all'interno dell'unità logica senza alcuna modifica.

Il settore in basso a destra della ALU contiene un sommatore per calcolare la somma di A e B, e gestire allo stesso tempo i riporti; la gestione dei riporti è necessaria dato che con ogni probabilità vari circuiti dello stesso tipo potrebbero essere collegati fra loro per permettere operazioni su intere parole. Circuiti come quello mostrato nella Figura 3.19 sono attualmente disponibili e vengono chiamati **bit slices** ("suddivisioni di un bit"), che permettono ai progettisti di calcolatori di costruire ALU di larghezza arbitraria. La Figura 3.20 mostra una ALU a 8 bit costruita unendo otto ALU a 1 bit. Il segnale INC è utilizzato solo per le operazioni di addizione; quando è attivo permette di incrementare il risultato (cioè sommargli 1), rendendo così possibile il calcolo di somme come $A + 1$ e $A + B + 1$.

3.2.4 Clock

In molti circuiti digitali l'ordine secondo il quale si verificano gli eventi è cruciale. In alcuni casi un evento deve precederne un altro, mentre in altre situazioni due eventi devono avvenire simultaneamente. Molti circuiti digitali utilizzano dei clock per gestire la sincronizzazione e permettere ai progettisti di ottenere le relazioni temporali desiderate. In questo contesto un **clock** è un circuito che emette una serie di impulsi di larghezza definita e a intervalli temporali costanti. L'intervallo temporale compreso tra le estremità di due impulsi consecutivi è detto **ciclo di clock**. La frequenza degli impulsi è generalmente compresa tra 1 e

500 MHz, corrispondenti a cicli di clock compresi tra 1000 e 2 ns. Per ottenere un'accuratezza elevata di solito la frequenza di clock è controllata da un oscillatore a cristalli.

In un calcolatore possono verificarsi più eventi durante uno stesso ciclo di clock. Se però è necessario che si verifichino in uno specifico ordine occorre dividere il ciclo di clock in sottocicli. Una tecnica che viene spesso utilizzata per ottenere una risoluzione più fine rispetto a quella del clock consiste nell'intercettare la linea del clock principale e inserirla in un circuito di cui si conosce il ritardo; in questo modo è possibile generare un secondo segnale di clock la cui fase è traslata rispetto a quella del clock principale, come mostra la Figura 3.21(a). Il diagramma di temporizzazione della Figura 3.21(b) fornisce quattro diversi riferimenti temporali utilizzabili per sincronizzare eventi discreti.

1. Fronte di salita di C1.
2. Fronte di discesa di C1.
3. Fronte di salita di C2.
4. Fronte di discesa di C2.

Associando diversi eventi ai quattro fronti è possibile stabilire per loro una desiderata sequenza. Se, all'interno di ogni ciclo di clock, sono necessari più di quattro riferimenti temporali occorre collegare al clock principale altre linee secondarie e utilizzare circuiti con ritardi diversi.

In alcuni circuiti si è interessati maggiormente agli intervalli temporali rispetto che agli istanti di tempo. A un evento potrebbe per esempio essere consentito di verificarsi in qualsiasi istante durante il quale C1 è alto e non per forza esattamente in corrispondenza del fronte di salita. Un altro evento potrebbe verificarsi solo quando C2 è alto. Se sono necessari più di due diversi intervalli temporali si possono utilizzare più linee di clock oppure si può fare in modo che gli intervalli in cui i segnali dei due clock sono alti si sovrappongano

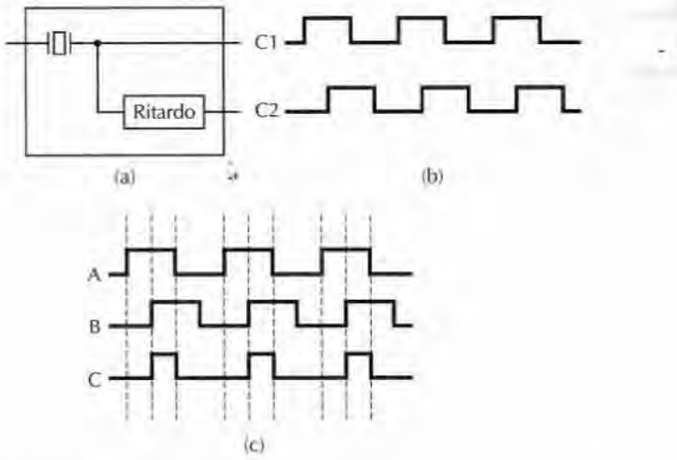


Figura 3.21 (a) Clock. (b) Diagramma di temporizzazione del clock. (c) Generazione di un clock asimmetrico.

parzialmente. In quest'ultimo caso si possono distinguere quattro diversi intervalli: $\overline{C1}$ AND $\overline{C2}$, $\overline{C1}$ AND C2, C1 AND $\overline{C2}$ e C1 AND C2.

Per essere più precisi i clock sono simmetrici nel senso che il tempo speso nello stato in cui il segnale è alto è uguale al tempo speso quando il segnale è basso, come mostra la Figura 3.21(b). Per generare una sequenza di impulsi asimmetrici il clock principale viene sfasato utilizzando un circuito ritardante e ne viene calcolato l'AND logico rispetto al segnale originale, come mostra il segnale C della Figura 3.21(c).

3.3 Memoria

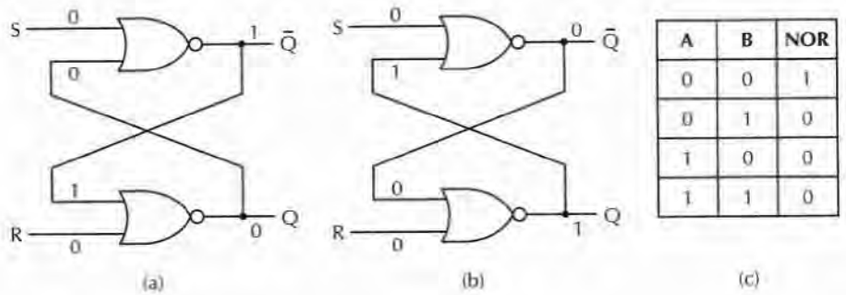
Una componente essenziale di ogni calcolatore è la memoria; se non ci fosse non potrebbe esistere nessun calcolatore, almeno nella forma in cui lo conosciamo. La memoria è utilizzata per conservare sia le istruzioni da eseguire sia i dati. Nei paragrafi successivi esamineremo i componenti base di un sistema di memoria; partiremo dal livello delle porte logiche per comprenderne il funzionamento e il modo in cui sono combinate per formare memorie più capaci.

3.3.1 Latch

Per creare una memoria a 1 bit è necessario disporre di un circuito che in qualche modo "ricordi" i precedenti valori di input. La Figura 3.22(a) mostra come sia possibile costruire un circuito di questo tipo utilizzando due porte NOR. Anche se è possibile costruire circuiti analoghi con porte NAND, nel seguito non li menzioneremo più, dato che dal punto di vista concettuale sono identici alle versioni basate su porte NOR.

Il circuito della Figura 3.22(a) è chiamato **latch SR** e ha due input: S, per impostare (*Setting*) il valore del latch e R per azzerarlo (*Resetting*). Il circuito ha anche due output, Q e $\overline{Q}$, che, come vedremo tra poco, sono complementari l'uno rispetto all'altro. Diversamente dalle reti combinatorie l'output di un latch non è determinato unicamente dai valori di input correnti.

Per vedere come ciò avviene assumiamo, come si verifica nella maggior parte dei casi, che sia S sia R valgano 0. Ai fini della spiegazione assumiamo inoltre che $Q = 0$. Dato che



A	B	NOR
0	0	1
0	1	0
1	0	0
1	1	0

(c)

Figura 3.22 (a) Latch di tipo NOR nello stato 0. (b) Latch di tipo NOR nello stato 1. (c) Tabella di verità del NOR.

Q viene reinserito come input della porta NOR visualizzata in alto, entrambi gli input di questa porta valgono 0 generando come output il valore 1. Il valore 1 è riutilizzato come input della porta visualizzata in basso, i cui input sono 1 e 0 e producono come output $Q = 0$. Questo stato, mostrato nella Figura 3.22(a), è stabile.

Immaginiamo ora che Q non sia 0, ma 1, mentre R e S continuano a valere 0. La porta logica visualizzata in alto ha come input 0 e 1; il suo output $\overline{Q}$ vale 0 ed è riutilizzato come input della porta visualizzata in basso. Anche questo stato è stabile ed è mostrato nella Figura 3.22(b). Uno stato in cui entrambi gli output valgono 0 è invece instabile, in quanto obbliga entrambe le porte ad avere due 0 come input, il che, se fosse vero, produrrebbe come risultato 1 e non 0. Analogamente è impossibile che entrambi gli output valgano 1, dato che ciò forzerebbe gli input a 0 e 1, il cui risultato dovrebbe essere 0 e non 1. La conclusione è semplice: per $R = S = 0$ il latch ha due stati stabili che identificheremo con 0 e 1 in base al valore di Q.

Esaminiamo ora quali effetti producono sullo stato del latch i valori dell'input. Supponiamo che S diventi 1, mentre $Q = 0$. Gli input del NOR superiore sono quindi 1 e 0 e forzano l'output $\overline{Q}$ a valere 0. Questo cambiamento porta entrambi gli input della porta inferiore a valere 0, producendo in output il valore 1. Settando quindi S (cioè impostandone il valore a 1) si può far passare lo stato da 0 a 1. Settando R a 1 quando il latch si trova nello stato 0 non si produce invece alcun effetto, dato che l'output della porta NOR in basso è 0 sia che i suoi input valgano 10 sia che valgano 11.

Applicando un ragionamento simile si può verificare facilmente che impostare S a 1 quando il latch è nello stato $Q = 1$ non produce alcun effetto, mentre settando R si modifica lo stato del latch portandolo nello stato $Q = 0$. Riassumendo si può dire che quando S è impostato temporaneamente a 1 lo stato del latch diventa $Q = 1$, indipendentemente dallo stato in cui si trovava precedentemente. Allo stesso modo quando si imposta temporaneamente R a 1 si forza il latch a passare nello stato $Q = 0$. Il circuito "ricorda" quindi quale valore, se S oppure R, è stato settato per ultimo; utilizzando questa proprietà è possibile costruire le memorie dei calcolatori.

Latch SR temporizzato

Spesso è preferibile impedire che un latch cambi di stato se non in specifici momenti. Un circuito che gode di questa caratteristica è detto **latch SR temporizzato**; per costruirlo occorre modificare leggermente il circuito visto precedentemente. Il circuito della Figura 3.23 ha un input aggiuntivo, il clock, il cui valore è generalmente 0. Quando il clock vale 0 en-

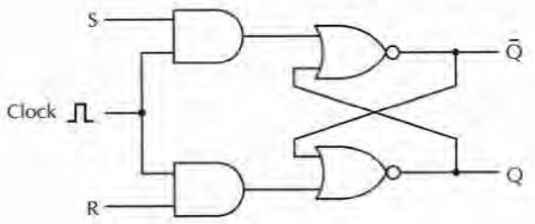


Figura 3.23 Latch SR temporizzato.

trambe le porte AND generano in output il valore 0, indipendentemente dai valori di S e R, impedendo quindi al latch di cambiare di stato. Quando il clock vale 1 le porte AND non bloccano più i segnali S e R che possono dunque tornare a pilotare lo stato del latch. Nonostante il suo nome non è obbligatorio che il segnale collegato all'input aggiuntivo debba per forza essere comandato da un clock. I termini ***enable*** e ***strobe*** sono largamente utilizzati per indicare che l'input clock vale 1, ovvero che il circuito è dipendente dallo stato di S e da quello di R.

Finora abbiamo intenzionalmente evitato di spiegare che cosa succede se sia S sia R valgono 1: il circuito diventa non deterministico finché sia R sia S non tornino ad assumere il valore 0. L'unico stato consistente per $S = R = 1$ è quando $Q = \overline{Q} = 0$; non appena entrambi gli input ritornano al valore 0 il latch deve tuttavia passare istantaneamente in uno dei suoi due stati stabili. Se uno dei due input torna a 0 prima dell'altro, prevale quello che rimane al valore 1 più a lungo, dato che quando uno solo degli input vale 1 esso determina lo stato del latch. Se invece entrambi gli input ritornano a 0 nello stesso istante (cosa che può verificarsi molto raramente) il latch passa in uno dei due stati stabili in modo del tutto casuale.

Latch D temporizzato

Un buon modo per risolvere l'ambiguità dei latch SR (causata dalla situazione $S = R = 1$) è... evitare che si verifichi. La Figura 3.24 mostra un circuito che ha un solo input, D. Dato che l'input della porta AND rappresentata in basso è sempre il complemento dell'input di quella superiore, non può mai accadere che entrambi gli input valgono 1. Quando $D = 1$ e il clock vale 1, il latch viene portato nello stato $Q = 1$, mentre, quando $D = 0$ e il clock vale 1, il latch passa nello stato $Q = 0$. In altre parole quando il clock vale 1 il valore corrente di D viene campionato e memorizzato nel latch. Questo circuito, chiamato **latch D temporizzato**, è una vera e propria memoria a 1 bit, in cui il valore memorizzato è sempre disponibile sulla linea Q. Per caricare in memoria il valore corrente di D occorre spedire un impulso positivo sulla linea del clock.

Il circuito appena descritto richiede 11 transistor; esistono tuttavia circuiti più sofisticati (ma con un'implementazione meno ovvia) che possono memorizzare 1 bit utilizzando solo sei transistor. Nelle implementazioni reali si adottano queste soluzioni più sofisticate, che impiegano meno transistor.

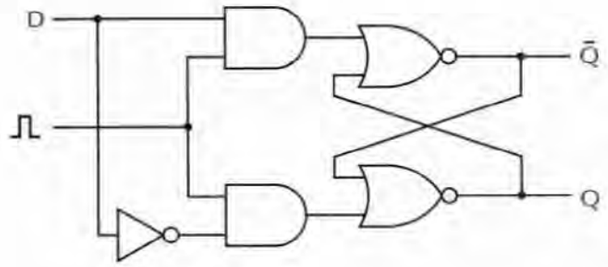


Figura 3.24 Latch D temporizzato.

3.3.2 Flip-flop

In molti circuiti è necessario campionare il valore di una certa linea in un particolare istante e memorizzarlo. In questi circuiti, chiamati **flip-flop**, la transizione di stato non si verifica quando il clock vale 1, ma durante la transizione del clock da 0 a 1 (fronte di salita) oppure da 1 a 0 (fronte di discesa). In questa situazione la lunghezza dell'impulso del clock non ha alcuna importanza, purché le transizioni si verifichino con sufficiente velocità.

Per sottolinearlo ulteriormente ripetiamo la differenza che c'è tra un flip-flop e un latch: un flip-flop è a **commutazione sul fronte**, mentre un latch è a **commutazione a livello**. Occorre però prestare attenzione al fatto che in letteratura spesso questi termini vengono confusi; molti autori utilizzano il termine "flip-flop" per indicare un latch e viceversa.

Esistono vari approcci per progettare un flip-flop. Se per esempio esistesse un metodo per generare un impulso di lunghezza estremamente breve sul fronte di salita, si potrebbe immettere tale impulso in un latch D. Il circuito che implementa questa soluzione è mostrato nella Figura 3.25(a).

A prima vista potrebbe sembrare che l'output della porta AND debba essere sempre zero, dato che l'AND tra un qualsiasi segnale e il suo inverso vale sempre zero; in realtà la situazione, anche se in modo sottile, è diversa. L'invertitore induce un piccolo, ma non nullo, ritardo di propagazione che permette al circuito di funzionare in modo corretto. Supponiamo di misurare la tensione in quattro punti diversi: a, b, c e d. Il segnale di input, misurato in a, è un lungo impulso di clock, come mostra, in basso, la Figura 3.25(b); il segnale in b è invece rappresentato dal grafico che nella figura si trova sopra quello precedente. Si noti che questo segnale, oltre a essere stato invertito, risulta anche sfasato di un ritardo che è di pochi nanosecondi (a seconda dal tipo di invertitore utilizzato).

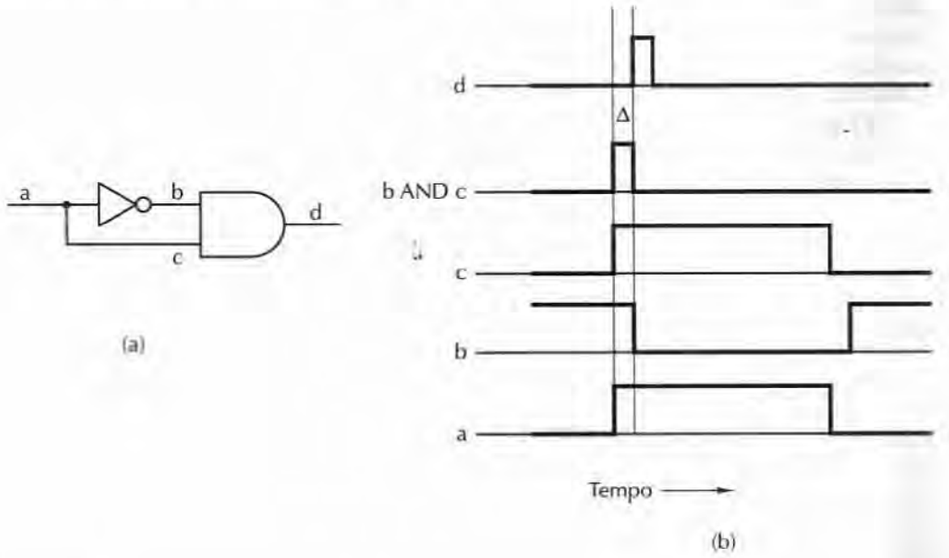


Figura 3.25 (a) Generatore d'impulsi. (b) Diagrammi temporali.

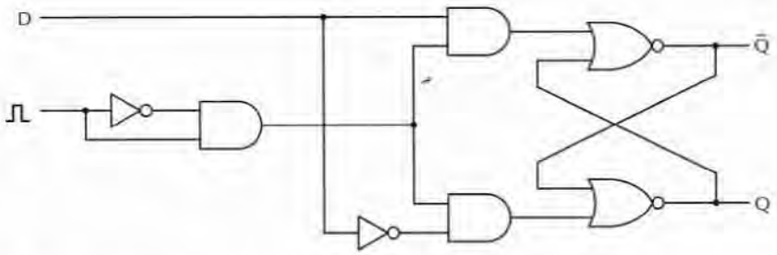


Figura 3.26 Flip-flop D.

Anche il segnale *c* subisce un ritardo, che però è determinato soltanto dal tempo di propagazione del segnale (a due terzi della velocità della luce). Se la distanza tra *a* e *c* è, per esempio, di 20 μm, il ritardo di propagazione è di un decimo di picosecondo. Tale valore è certamente trascurabile rispetto al tempo di commutazione dell'invertitore. Di conseguenza per qualsiasi intento o scopo del circuito, il segnale *c* può essere considerato identico al segnale *a*.

Quando gli input *b* e *c* vengono processati dalla porta AND si ottiene come risultato un impulso di breve durata, come mostra la Figura 3.25(b); la larghezza Δ dell'impulso è uguale al ritardo dell'invertitore, generalmente inferiore a 5 ns. Com'è illustrato nella Figura 3.25(b) l'output della porta AND corrisponde semplicemente a questo impulso sfasato del ritardo interno alla porta logica AND. Questo sfasamento temporale significa che il latch D verrà attivato con un ritardo fisso rispetto al fronte di salita del clock; tuttavia ciò non ha alcun effetto sulla durata dell'impulso. Un impulso di 5 ns che segnala quando campionare la linea D può essere considerato sufficientemente corto per una memoria con un ciclo della durata di 50 ns; in un caso simile il circuito completo potrebbe essere quello della Figura 3.26. Occorre sottolineare che il vantaggio dell'architettura di questo flip-flop è di essere di facile comprensione; tuttavia nella realtà si ricorre di solito ad architetture più sofisticate.

La Figura 3.27 mostra i simboli comunemente usati per i latch e i flip-flop. La Figura 3.27(a) indica un latch il cui stato viene caricato quando il clock, *CK*, vale 1; al contrario lo stato del latch della Figura 3.27(b) è generalmente 1, ma passa temporaneamente al valore 0 per assumere lo stato dalla linea *D*. Le Figure 3.27(c) e (d) rappresentano due flip-flop invece che due latch; ciò è indicato dal simbolo triangolare vicino all'input del clock. La Figura 3.27(c) cambia stato in corrispondenza del fronte di salita dell'impulso del clock (transizione da 0 a 1), mentre la Figura 3.27(d) cambia stato in corrispondenza del fronte di discesa (transizione da 1 a 0). Molti latch e flip-flop, seppur non tutti, hanno anche

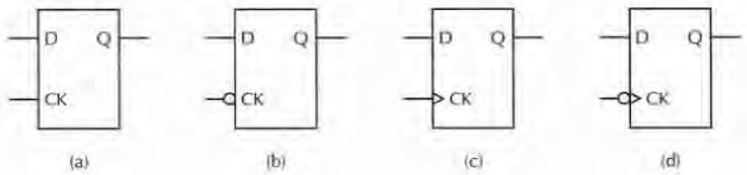


Figura 3.27 Flip-flop D temporizzati.

l'output $\overline{Q}$ e alcuni sono dotati di due input aggiuntivi, *Set* o *Preset* (che forzano lo stato a $Q = 1$) e *Reset* o *Clear* (che forzano lo stato a $Q = 0$).

3.3.3 Registri

Tra le svariate configurazioni in cui i flip-flop sono disponibili, la Figura 3.28(a) ne mostra una particolarmente semplice consistente in due flip-flop dotati dei segnali *Clear* e *Preset*. I due flip-flop, pur essendo integrati nello stesso chip a 14 pin, non sono in relazione l'uno

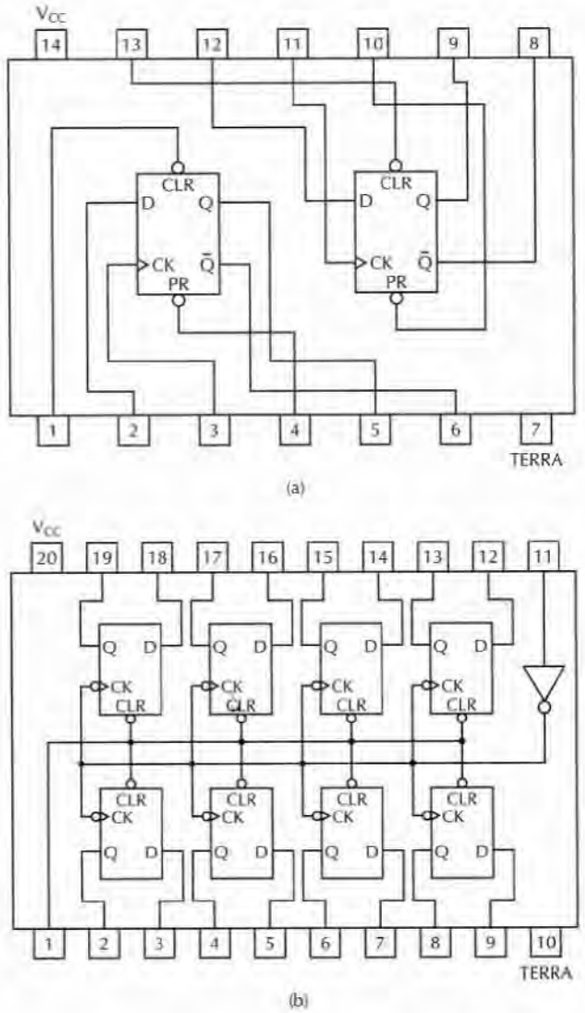


Figura 3.28 (a) Flip-flop D duale. (b) Flip-flop ottale.

con l'altro. La Figura 3.28(b) mostra invece una configurazione diversa: un flip-flop ottale. In questo caso gli otto flip-flop D (ai quali si deve il termine "ottale") non solo sono sprovvisti di $\overline{Q}$ e delle linee di reimpostazione, ma inoltre tutte le linee di clock sono collegate fra loro e pilotate dal pin 11. I singoli flip-flop sono del tipo mostrato nella Figura 3.27(d), anche se il cerchietto d'inversione non è presente per via dell'invertitore collegato al pin 11; per questa differenza i flip-flop vengono caricati in corrispondenza della transizione di salita. Anche tutti e otto i canali di cancellazione sono collegati fra loro, di modo che il passaggio del pin 1 al valore 0 forzi tutti i flip-flop ad assumere lo stato 0. Ci si potrebbe chiedere per quale motivo il pin 11 è invertito nella linea di input per poi essere invertito nuovamente in corrispondenza di ciascun segnale CK. Il motivo è che un segnale potrebbe non avere sufficiente corrente per pilotare tutti i flip-flop; l'invertitore svolge in realtà il ruolo di amplificatore.

Una ragione per unire tutte le linee di clock e di cancellazione potrebbe essere quella di risparmiare pin; in realtà in questa configurazione il vero motivo è che il chip è utilizzato come un registro a 8 bit e non come otto flip-flop scollegati. Unendo i rispettivi pin 1 e 11 è possibile utilizzare due di questi chip in parallelo per formare un registro a 16 bit. Nel corso del Capitolo 4 analizzeremo più da vicino i registri e il loro impiego.

3.3.4 Organizzazione della memoria

Nei paragrafi precedenti siamo partiti dalla semplice memoria a 1 bit della Figura 3.24 per arrivare alla memoria a 8 bit della Figura 3.28. Per realizzare memorie di dimensione maggiore è però necessaria un'organizzazione di tipo diverso, nella quale sia possibile indirizzare singole parole. La Figura 3.29 mostra un'organizzazione della memoria molto comune che soddisfa questo requisito. L'esempio illustra una memoria con quattro parole a 3 bit in cui ciascuna operazione legge o scrive un'intera parola. Anche se la memoria ha una capacità totale (12 bit) decisamente maggiore rispetto a quella del flip-flop ottale visto precedentemente, essa richiede un numero inferiore di pin. Una caratteristica ancora più rilevante è che questa organizzazione è facilmente estendibile a memorie di dimensione maggiore.

A prima vista la memoria della Figura 3.29 potrebbe sembrare complessa, ma in realtà, se si considera la regolarità della struttura, si nota che è piuttosto semplice. Delle otto linee di input tre sono per i dati, I_0, I_1 e I_2, due per l'indirizzo, A_0 e A_1, e tre per i controlli, CS per la selezione del chip, RD per distinguere tra lettura e scrittura e OE per l'abilitazione dell'output. Le tre linee di output, O_0, O_1 e O_2, sono invece dedicate ai dati. In linea di principio è possibile inserire questa memoria in un contenitore con 14 pin, alimentazione e terra compresi, contro i 20 pin di un flip-flop ottale.

Per utilizzare questo chip è necessario che una componente logica esterna imposti, in caso di lettura, sia CS sia RD al valore alto (valore logico 1), mentre, in caso di scrittura, li imposti al valore basso (valore logico 0). Le due linee dell'indirizzo devono essere impostate in modo da indicare quale delle quattro parole a 3 bit deve essere letta o scritta. In lettura le linee di input non vengono utilizzate e la parola selezionata viene resa disponibile sulle linee di output. In scrittura i bit presenti sulle linee di input dei dati vengono caricati nella parola di memoria selezionata, mentre le linee di output non vengono utilizzate.

Per comprendere il funzionamento della memoria osserviamo ora in modo più dettagliato la Figura 3.29. Le quattro porte AND situate a sinistra della memoria formano un decodificatore e servono a selezionare la parola. I quattro invertitori in input sono stati

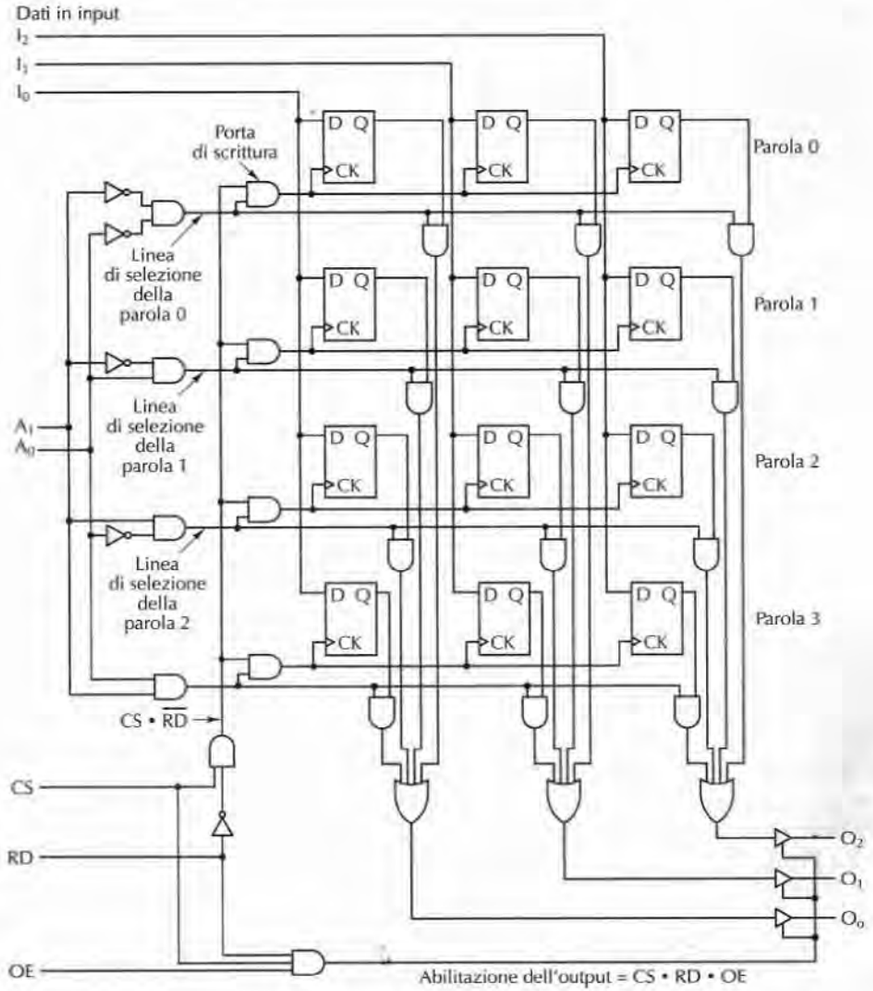


Figura 3.29 Diagramma logico di una memoria 4 × 3. Ogni riga è una della quattro parole a 3 bit. Lettura e scrittura riguardano sempre parole complete.

collocati per far sì che ciascuna porta logica sia abilitata (l'output è alto) da un indirizzo diverso. Ciascuna porta è collegata a una linea per la selezione di una delle parole, indicate dall'alto verso il basso con i numeri 0, 1, 2 e 3. Quando il chip viene selezionato per un'operazione di scrittura la linea verticale etichettata con CS · $\overline{RD}$ assume valore alto, abilitando quindi una della quattro porte di scrittura; la porta abilitata dipende da quale linea per la selezione della parola ha assunto valore alto. L'output della porta di scrittura guida tutti i segnali CK relativi alla parola selezionata, caricando i dati di input nei flip-flop che

costituiscono la parola stessa. La scrittura è possibile soltanto quando CS è alto e RD è basso; l'unica parola a essere scritta è quella selezionata dai segnali A_0 e A_1, mentre le altre non vengono modificate.

L'operazione di lettura è simile: la decodifica dell'indirizzo, per esempio, si svolge esattamente nello stesso modo, la linea CS · $\overline{\text{RD}}$ assume però valore basso, disabilitando di conseguenza tutte le porte di scrittura e impedendo la modifica dei flip-flop. La linea per la selezione della parola scelta abilita le porte AND cui sono collegati i bit Q della parola selezionata. La parola selezionata spedisce quindi i propri dati alle porte OR mostrate nella parte bassa della figura. Dato che le altre parole generano in output valori 0, il risultato delle porte OR è identico al valore memorizzato nella parola selezionata. Le tre parole non selezionate non forniscono quindi alcun contributo all'output finale.

Anche se si potrebbe progettare un circuito nel quale le tre porte OR siano collegate direttamente alle tre linee di output dei dati, in alcuni casi ciò potrebbe causare dei problemi. Nell'illustrazione abbiamo distinto le linee di input da quelle di output, ma nelle memorie reali si utilizzano le stesse linee. Se avessimo collegato le porte OR alle linee di output il chip avrebbe cercato di spedire in output i dati anche durante un'operazione di scrittura, forzando ciascuna linea ad assumere un particolare valore e interferendo quindi con i dati di input. Per questo motivo è preferibile avere la possibilità di connettere le porte OR alle linee di output durante le letture e di disconnetterle completamente durante le scritture. Quello di cui abbiamo bisogno è un interruttore elettronico che possa instaurare o interrompere una connessione in pochi nanosecondi.

Un interruttore del genere è chiamato **buffer non invertente** e la Figura 3.30(a) mostra il simbolo che lo rappresenta. Esso ha un dato di input, un dato di output e un input di controllo. Quando l'input di controllo è alto il buffer funge da collegamento, come mostra la Figura 3.30(b). Quando invece l'input di controllo è basso il buffer si comporta come un circuito aperto, come mostra la Figura 3.30(c): è come se qualcuno avesse scollegato il dato di output dal resto del circuito con una pinza tagliafili. Tuttavia, diversamente dall'utilizzo di una pinza tagliafili, la connessione può essere ripristinata in pochi nanosecondi semplicemente reimpostando il segnale di controllo al valore alto.

La Figura 3.30(d) mostra un **buffer invertente** che si comporta come un normale invertitore quando il controllo ha valore alto e disconnette l'output dal circuito quando il controllo ha valore basso. I due buffer sono **dispositivi a tre stati**, in quanto possono generare in output i valori 0 e 1, oppure nessuno dei due (circuito aperto). I buffer inoltre amplificano il segnale e possono quindi guidare più input allo stesso tempo; talvolta vengono

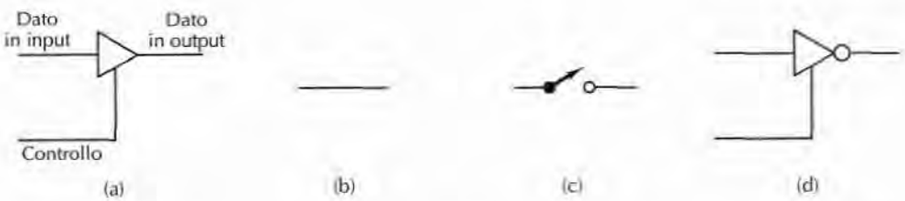


Figura 3.30 (a) Buffer non invertente. (b) Risultato di (a) quando il controllo è alto. (c) Risultato di (a) quando il controllo è basso. (d) Buffer invertente.

impiegati all'interno dei circuiti proprio per questa ragione, anche quando non sono richieste le loro proprietà d'inversione.

Tornando al circuito della memoria dovrebbe essere chiaro quale sia il ruolo dei buffer non invertenti che si trovano sulle linee di output dei dati. Quando CS, RD e OE hanno tutti il valore alto anche il segnale per l'abilitazione dell'output è alto; ciò permette di attivare i buffer e di spedire una parola sulle linee di output. Al contrario, quando uno qualsiasi dei segnali CS, RD e OE è basso, l'output è scollegato dal resto del circuito.

3.3.5 Chip di memoria

Una proprietà interessante della memoria della Figura 3.29 è che può essere facilmente ampliata. Quella che abbiamo descritto è una memoria 4 × 3, costituita cioè da quattro parole di 3 bit ciascuna. Per estenderla alla dimensione 4 × 8 occorre semplicemente aggiungere cinque colonne composte da quattro flip-flop ciascuna, così come cinque linee di input aggiuntive e altrettante linee di output. Per passare dalla dimensione 4 × 3 a quella 8 × 3 si devono aggiungere quattro nuove righe di tre flip-flop ciascuna, oltre a una linea aggiuntiva, A_2, per l'indirizzo. Una memoria con questa struttura dovrebbe avere un numero di parole esprimibile come una potenza di 2 al fine di massimizzare l'efficienza; il numero di bit di una parola può invece assumere qualsiasi valore.

Dato che la tecnologia dei circuiti integrati è particolarmente adatta a realizzare chip la cui struttura interna abbia uno schema bidimensionale ripetuto, i chip di memoria ne sono un'applicazione ideale. Con l'avanzamento della tecnologia continua ad aumentare il numero di bit che si può inserire in un chip. Tale valore raddoppia all'incirca ogni 18 mesi (legge di Moore). I chip di dimensioni maggiori non sempre rendono obsoleti quelli più piccoli, dato che spesso occorre trovare dei compromessi tra diversi fattori, quali la capacità, la velocità, l'alimentazione, il prezzo e la comodità di interfacciamento. Di solito i chip di dimensioni maggiori sono venduti come prodotti di qualità superiore e quindi a un prezzo per bit più elevato rispetto a quelli più vecchi e più piccoli.

Fissata una dimensione della memoria esistono diversi modi per organizzare il chip. La Figura 3.31 mostra due possibili organizzazioni per un vecchio chip di memoria della dimensione di 4 Mbit: 512K × 8 e 4096K × 1. Per inciso, le dimensioni dei chip di memoria sono generalmente indicate in bit, e non in byte, e anche noi ci atterremo a questa convenzione. Nella Figura 3.31(a) sono necessarie 19 linee per indirizzare uno dei 2^{19} byte e otto linee di output per caricare e memorizzare i byte selezionati.

Occorre sottolineare un aspetto della terminologia utilizzata. Su alcuni pin l'applicazione di un'alta tensione genera una qualche azione, mentre su altri le azioni sono innescate da bassi valori di tensione. Per evitare confusione diremo che un segnale è **asserito** (piuttosto che dire che assume valore alto o basso) per indicare che è impostato in modo da generare una qualche azione. Alcuni pin sono asseriti con valore alto, mentre altri con valore basso. I pin asseriti con valore basso sono identificati da una linea sopra il loro nome. Un segnale chiamato CS (*Chip Select*) viene quindi asserito con il valore alto, mentre uno chiamato $\overline{\text{CS}}$ è asserito con il valore basso. L'opposto di asserito è **negato**; quando non succede nulla di particolare i pin sono negati.

Torniamo ora al nostro chip di memoria. Dato che un calcolatore ha di solito vari chip di memoria è necessario disporre di un segnale che selezioni il chip richiesto in un dato momento, di modo che solo esso risponda al segnale, mentre tutti gli altri lo ignorino. Per que-

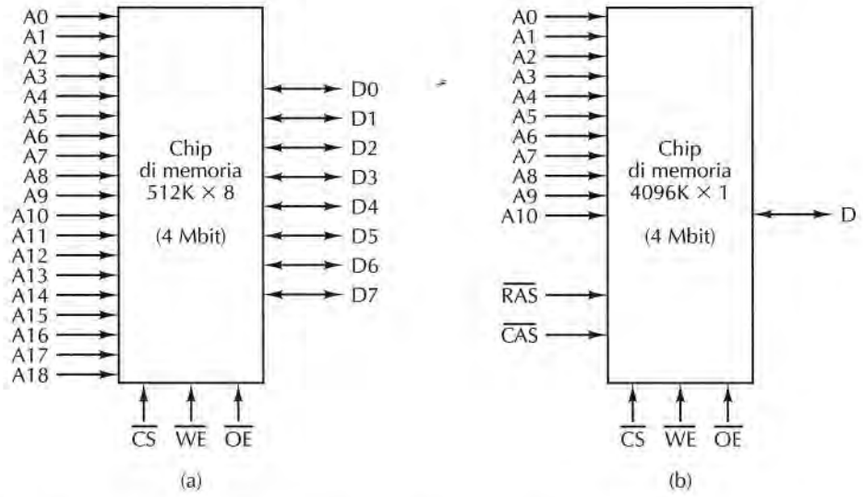


Figura 3.31 Due modi di organizzare un chip di memoria a 4 Mbit.

sto scopo si utilizza il segnale $\overline{\text{CS}}$, asserito quando si intende abilitare il chip. Inoltre è necessario un metodo per distinguere tra le operazioni di lettura e quelle di scrittura; ciò viene fatto dal segnale $\overline{\text{WE}}$ (*Write Enable*), utilizzato per indicare che i dati devono essere scritti piuttosto che letti. Infine il segnale $\overline{\text{OE}}$ (*Output Enable*) viene asserito per guidare i segnali di output; quando non è asserito, l'output del chip è sconnesso dal circuito.

Nella Figura 3.31(b) si utilizza un diverso schema per l'indirizzamento. Al suo interno questo chip è organizzato come una matrice di 2048 × 2048 celle a 1 bit, che forniscono una capacità totale di 4 Mbit. Per indirizzare il chip si seleziona inizialmente una riga immettendo un numero a 11 bit sui pin dell'indirizzo e asserendo il segnale $\overline{\text{RAS}}$ (*Row Address Strobe*, "strobe dell'indirizzo di riga"). Successivamente si immette sui pin dell'indirizzo un numero di colonna e si asserisce il segnale $\overline{\text{CAS}}$ (*Column Address Strobe*, "strobe dell'indirizzo di colonna"). Il chip risponde accettando, oppure generando in output, un dato da 1 bit.

I chip di memoria di grandi dimensioni sono spesso costruiti come matrici di $n \times n$ indirizzate da numeri di riga e colonna. Questo tipo di architettura riduce il numero di pin necessari, ma rende allo stesso tempo più lento il chip, in quanto sono necessari due cicli di indirizzamento, uno per la riga e uno per la colonna. Per riguadagnare parte della velocità persa a causa dell'architettura, in alcuni chip è possibile specificare un indirizzo di riga seguito da una sequenza di indirizzi di colonna in modo da poter accedere a bit consecutivi all'interno di una stessa riga.

Anni fa i chip di memoria più grandi erano spesso organizzati come nella Figura 3.31(b). Quando però la dimensione delle parole di memoria è passata da 8 bit a 32 bit, i chip larghi 1 bit sono diventati poco convenienti. Per costruire una memoria con parole a 32 bit mediante chip 4096K × 1 è infatti necessario utilizzarne 32 in parallelo.

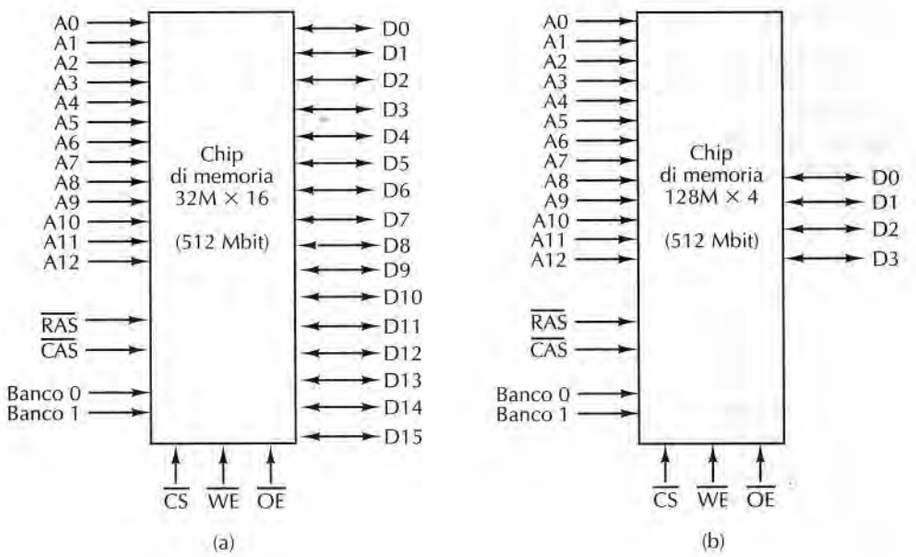


Figura 3.32 Due modi di organizzare un chip di memoria a 512 Mbit.

Questi 32 chip forniscono una capacità totale di 16 MB, mentre l'utilizzo di chip 512K × 8 richiede solo quattro chip in parallelo e fornisce memorie della dimensione di 2 MB. Per evitare di avere 32 chip per la memoria, quasi tutti i produttori offrono attualmente famiglie di chip con larghezze da 4, 8 e 16 bit. Ovviamente se si considerano parole a 64 bit la situazione è ancora più critica.

La Figura 3.32 mostra due esempi di moderni chip a 512 Mbit. I chip hanno quattro banchi di memoria di 128 Mbit ciascuno e richiedono quindi due linee per la selezione del banco desiderato. La Figura 3.32(a) mostra un progetto 32M × 16, con 13 linee per il segnale $\overline{\text{RAS}}$, 10 linee per il segnale $\overline{\text{CAS}}$ e 2 linee per la selezione del banco. L'insieme di questi 25 segnali permette di indirizzare ciascuna delle 2^{25} celle interne, tutte a 16 bit. La Figura 3.32(b) mostra invece un progetto 128M × 4, con 13 linee per il segnale $\overline{\text{RAS}}$, 12 linee per il segnale $\overline{\text{CAS}}$ e 2 linee per la selezione del banco. In questo caso i 27 segnali permettono di selezionare una qualsiasi delle 2^{27} celle interne a 4 bit. La scelta del numero di righe e colonne di un chip dipende da scelte ingegneristiche; in ogni caso non è necessario che la matrice sia quadrata.

Questi esempi mettono in evidenza due distinti problemi, non collegati tra loro, che riguardano la progettazione dei chip di memoria. Il primo concerne la larghezza dell'output (in bit): il chip invia 1, 4, 8, 16 o un altro numero di bit in una volta sola? Il secondo corrisponde a chiedersi se tutti gli indirizzi vengono forniti allo stesso tempo su pin distinti oppure si forniscono prima i valori di riga e poi quelli di colonna come negli esempi della Figura 3.32. Un ingegnere che deve costruire un chip di memoria, prima di cominciare a progettare il proprio chip, deve dare una risposta a queste domande.

3.3.6 RAM e ROM

Tutte le memorie studiate finora possono essere sia lette sia scritte. Le memorie di questo tipo sono chiamate **RAM** (*Random Access Memory*, "memoria ad accesso casuale"); il termine è fuorviante in quanto tutti i chip di memoria sono accessibili in modo casuale, ma ormai è talmente radicato che è impossibile modificarlo. Esistono due tipi di RAM: statiche e dinamiche. Le **RAM statiche** (SRAM) sono costruite utilizzando circuiti simili ai flip-flop D e hanno la proprietà di mantenere il proprio contenuto fintanto che vi è alimentazione: per secondi, minuti, ore o anche giorni. Le RAM statiche sono molto veloci e i loro tempi di accesso sono usualmente dell'ordine dei nanosecondi. Per questa ragione sono molto diffuse come memorie cache di secondo livello.

Al contrario **le RAM dinamiche** (DRAM) non usano flip-flop, ma sono composte da un array di celle, ciascuna delle quali contiene un transistor e un piccolo condensatore. Il condensatore può essere caricato o scaricato per memorizzare i valori 0 oppure 1. Dato che la carica elettrica tende a disperdersi occorre effettuare, a intervalli temporali di pochi millisecondi, un ***refresh*** (ricarica) di ciascun bit della RAM dinamica per evitare che i dati vadano persi. Le RAM dinamiche richiedono un'interfaccia più complessa rispetto a quelle statiche per via dei componenti logici esterni che devono occuparsi del refresh; in molte applicazioni questo svantaggio è compensato dalla maggior capacità che le RAM dinamiche possono offrire.

Le RAM dinamiche hanno un'elevata densità (molti bit per chip) dato che richiedono soltanto un transistor e un condensatore per bit (rispetto ai sei transistor per bit della migliore RAM statica). Per questa ragione le memorie centrali sono quasi sempre costruite utilizzando RAM dinamiche. Questa grande capacità ha però un prezzo: le RAM dinamiche sono lente (decine di nanosecondi). Per cercare di trarre vantaggio dalle proprietà di entrambi i tipi di memoria spesso si ricorre a un uso combinato, realizzando una cache con RAM statica e la memoria centrale con RAM dinamica.

Esistono vari tipi di chip di RAM dinamica. Il più datato è il DRAM **FPM** (*Fast Page Mode*), ancora utilizzato nei calcolatori più vecchi. Internamente è organizzato a matrice di bit, e per funzionare occorre che l'hardware fornisca prima l'indirizzo di riga e poi quello di colonna, allo stesso modo di quanto avveniva con i segnali $\overline{\text{RAS}}$ e $\overline{\text{CAS}}$ nel contesto della Figura 3.31. Inoltre si utilizzano dei segnali espliciti per indicare alla memoria quando rispondere; dunque il funzionamento della memoria è asincrono rispetto al clock principale del sistema.

La DRAM FPM è stata sostituita dalla DRAM **EDO** (*Extended Data Output*) in cui un riferimento alla memoria può avere inizio ancor prima che sia completato il precedente. Questa semplice strategia a pipeline non accelera il singolo riferimento, ma aumenta la larghezza di banda della memoria.

I chip FPM e EDO lavoravano in modo accettabile quando i chip di memoria avevano dei cicli di 12 ns, o erano addirittura più lenti. Quando i processori sono diventati così veloci da richiedere memorie più rapide, i chip FPM e EDO sono stati sostituiti dalle **SDRAM** (*Synchronus DRAM*, DRAM sincrona); la SDRAM è una RAM ibrida, in parte statica e in parte dinamica, ed è guidata dal clock principale del sistema. Il maggior vantaggio di queste RAM è che il clock elimina la necessità dei segnali di controllo per specificare al chip quando deve rispondere. Al contrario la CPU comunica alla memoria per quanti cicli deve funzionare e poi fa partire l'esecuzione. Ad ogni ciclo la memoria manda in output 4, 8 o 16 bit, a seconda del numero delle sue linee di output. L'eliminazione dei segnali di controllo aumenta il tasso di trasferimento dati tra CPU e memoria.

Dopo la SDRAM il passaggio successivo è stata la SDRAM **DDR** (*Double Data Rate*, "memoria a duplice tasso di trasferimento"). In queste memorie il chip produce un output sul fronte di salita del segnale di clock e uno sul fronte di discesa, raddoppiando così il tasso di trasferimento dati. Un chip DDR largo 8 bit e funzionante a 200 MHz genera in output, per 200 milioni di volte al secondo (ovviamente per un breve intervallo di tempo), due valori a 8 bit; questi dati corrispondono a un *burst rate* (cioè una frequenza di picco) teorico di 3,2 Gbps.

Chip di memoria non volatile

Le RAM non sono l'unico tipo di chip di memoria. In molte applicazioni, come i giocattoli, gli elettrodomestici e le automobili, il programma e alcuni dati devono rimanere memorizzati anche quando viene tolta l'alimentazione. Inoltre non si richiede mai la modifica del programma né dei dati installati. Questi requisiti hanno portato allo sviluppo delle **ROM** (*Read-Only Memory*, "memoria di sola lettura") che non possono essere modificate o cancellate, né intenzionalmente né accidentalmente. I dati sono inseriti durante la sua fabbricazione. Ciò avviene esponendo alla luce un materiale fotosensibile attraverso una maschera contenente il pattern di bit desiderato e incidendo quindi la superficie esposta (o non esposta). L'unico modo per cambiare il programma consiste nella sostituzione dell'intero chip.

In grandi volumi le ROM sono molto più economiche delle RAM dato che il costo per la realizzazione della maschera viene ammortizzato dal gran numero di esemplari. Le memorie ROM non sono però flessibili, dato che non possono essere modificate dopo la fabbricazione; inoltre il tempo che passa tra l'ordine e la ricezione delle ROM può essere di settimane. Per far sì che le aziende potessero sviluppare più agevolmente nuovi prodotti basati su ROM sono state inventate le **PROM** (*Programmable ROM*, ROM programmabile). Una PROM differisce da una ROM per il fatto che può essere programmata (una volta) sul posto, evitando quindi i tempi per il completamento dell'ordine da parte del produttore. Molte PROM contengono un array di piccoli fusibili, che possono essere bruciati: ciò viene fatto selezionando righe e colonne e applicando un'alta tensione a un particolare pin del chip.

Il passo successivo in questa linea di prodotti è stata la **EPROM** (*Erasable PROM*, PROM cancellabile), i cui campi non solo possono essere programmati, ma anche cancellati. Quando si espone la piccola lente al quarzo che si trova nella EPROM a un'intensa luce ultravioletta per 15 minuti, tutti i bit assumono il valore 1. Se ci si aspetta che la produzione di un chip richiederà un gran numero di fasi di test le EPROM risultano molto più economiche delle PROM, dato che possono essere riutilizzate. In genere le EPROM hanno la stessa organizzazione delle RAM statiche. La EPROM 27C040 a 4 Mbit utilizza per esempio la stessa organizzazione della Figura 3.31(a), generalmente impiegata per una RAM statica.

Meglio ancora delle EPROM sono la **EEPROM** (la prima E sta per *elettricamente*) che possono essere cancellate applicando impulsi elettrici invece di dover inserire il chip in una camera speciale per l'esposizione alla luce ultravioletta. Inoltre queste memorie sono riprogrammabili senza doverle rimuovere dal circuito, mentre una EPROM deve essere inserita nello speciale dispositivo che ne permette la programmazione. Le EEPROM però non possono essere più grandi di 1/64 delle comuni EPROM e la loro velocità è solo la metà. Le EEPROM non possono competere con le DRAM e le SRAM dato che sono 10 volte più lente, hanno una capacità 100 volte minore e sono molto più costose; esse sono utilizzate solo quando la loro proprietà di non volatilità è cruciale.

Tipo	Categoria	Cancellazione	Byte modificabili	Volatile	Tipico utilizzo
SRAM	Read/write	Elettrica	Sì	Sì	Cache di secondo livello
DRAM	Read/write	Elettrica	Sì	Sì	Memoria centrale (vecchia)
SDRAM	Read/write	Elettrica	Sì	Sì	Memoria centrale (recente)
ROM	Read-only	Impossibile	No	No	Elettrodomestici (prodotti in grandi volumi)
PROM	Read-only	Impossibile	No	No	Dispositivi (prodotti in piccoli volumi)
EPROM	Read-mostly	Raggi UV	No	No	Prototipazione di dispositivi
EEPROM	Read-mostly	Elettrica	Sì	No	Prototipazione di dispositivi
Flash	Read/write	Elettrica	No	No	"Pellicola" per macchine fotografiche digitali

Figura 3.33 Confronto tra vari tipi di memoria.

Un tipo più recente di EEPROM è la **memoria flash**. Diversamente dalla EPROM, che è cancellabile per esposizione alla luce ultravioletta, e dalla EEPROM, nella quale ogni singolo byte è cancellabile, la memoria flash è cancellabile a blocchi e riscrivibile. Come la EEPROM anche la memoria flash può essere cancellata senza doverla rimuovere dal circuito. Vari produttori forniscono piccole schede con circuiti stampati che contengono una memoria flash; la sua capacità può raggiungere 1 GB e spesso viene usata, oltre che per altri scopi, come "pellicola" per memorizzare le foto delle macchine fotografiche digitali. In futuro le memorie flash potrebbero sostituire i dischi; ciò rappresenterebbe un enorme miglioramento per via del loro tempo di accesso di 50 ns. Il principale problema ingegneristico è che attualmente si esauriscono dopo 100.000 cancellazioni, mentre i dischi resistono per vari anni, indipendentemente da quante volte vengono riscritti. La Figura 3.33 mostra un riepilogo dei vari tipi di memorie.

3.4 Chip della CPU e bus

Forti delle conoscenze acquisite sui chip SSI, MSI e su quelli di memoria possiamo ora cominciare a mettere insieme i vari pezzi per affrontare il sistema nel suo complesso. In questo paragrafo analizzeremo inizialmente alcuni aspetti generali delle CPU dal punto di vista del livello logico digitale, **contatti** compresi (significato dei segnali sui singoli pin), e forniremo anche un'introduzione all'architettura dei bus, dato che le CPU dipendono strettamente dal modo in cui questi sono progettati. Nei paragrafi successivi daremo esempi più dettagliati di CPU, bus, e loro interfacce.

3.4.1 Chip della CPU

Tutte le CPU moderne sono contenute in un unico chip, rendendo in questo modo ben definita l'interazione con il resto del sistema. Ogni chip di CPU ha un insieme di pin attraverso il quale passano tutte le relative comunicazioni verso il mondo esterno. Alcuni pin spediscono i segnali della CPU, altri ricevono quelli del mondo esterno e altri ancora possono fare entrambe le cose. Conoscendo la funzione di tutti i pin possiamo capire in quale modo la CPU interagisce con la memoria e i dispositivi di I/O nel livello logico digitale.

I pin di una CPU possono essere divisi in tre tipi: indirizzi, dati e controlli. Questi pin sono collegati ad analoghi pin presenti sulla memoria, o sui chip di I/O, mediante un insieme di cavi paralleli chiamato bus. La CPU, per prelevare un'istruzione, ne imposta l'indirizzo sui suoi pin di indirizzamento, e poi asserisce una o più linee di controllo per informare la memoria che vuole leggere una parola. La memoria risponde spedendo la parola richiesta sui pin della CPU dedicati ai dati e asserendo un segnale che notifica che l'operazione è stata completata. Quando la CPU vede questo segnale accetta la parola ed esegue l'istruzione.

Se l'istruzione richiede la lettura o la scrittura di certi dati, l'intero processo viene ripetuto per ogni parola di dati. In seguito entreremo nei dettagli, ma per il momento è importante capire che la CPU comunica con la memoria e i dispositivi di I/O inviando e accettando segnali sui propri pin. Non esiste altro tipo di comunicazione.

Due dei principali parametri che determinano le prestazioni di una CPU sono il numero di pin d'indirizzo e il numero di pin di dati. Un chip con m pin d'indirizzo può indirizzare fino a 2^m locazioni di memoria; valori comuni di m sono 16, 20, 32 e 64. Analogamente un chip con n pin di dati può leggere o scrivere in un'unica operazione una parola a n bit (valori comuni di n sono 8, 16, 32, 36 e 64). Una CPU con 8 pin di dati dovrà eseguire quattro operazioni per leggere una parola a 32 bit, mentre una CPU con 32 pin di dati può compiere lo stesso lavoro in una sola operazione. Il chip con 32 pin di dati è quindi molto più veloce, ma molto più costoso.

Oltre ai pin d'indirizzo e dei dati, le CPU sono dotate anche di alcuni pin di controllo. Questi regolano il flusso e la temporizzazione dei dati da e verso la CPU, e possono essere utilizzati anche in vari altri modi. Tutte le CPU hanno pin per l'alimentazione (generalmente +3,3 oppure +5 volt) e per la terra, oltre a un segnale di clock (un'onda quadra con una particolare frequenza); gli altri pin variano invece in modo considerevole da chip a chip. Ciononostante i pin di controllo possono essere approssimativamente raggruppati nelle seguenti categorie principali:

1. controllo del bus
2. interrupt
3. arbitraggio del bus
4. comunicazione con il coprocessore
5. stato
6. altro.

In seguito descriveremo brevemente queste categorie, mentre forniremo maggiori dettagli quando studieremo i chip del Pentium 4, dell'UltraSPARC III e dell'8051. La Figura 3.34 mostra un generico chip di una CPU che usa questi gruppi di segnali.

I pin per il controllo del bus mandano principalmente sul bus dei segnali dalla CPU (destinati alla memoria e ai chip di I/O) per notificare quando la CPU vuole leggere dalla

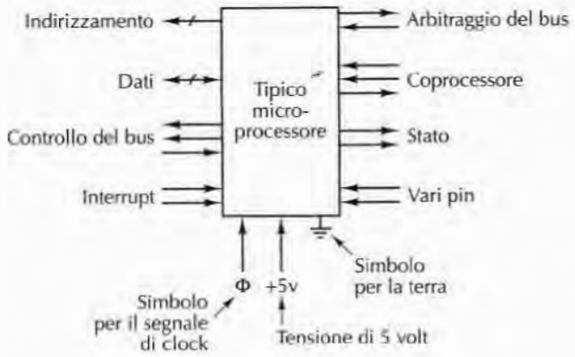


Figura 3.34 Una generica CPU. Le frecce indicano i segnali di input e di output. I trattini diagonali indicano pin multipli; per una specifica CPU un valore ne indica la quantità.

o scrivere in memoria oppure compiere altre azioni. La CPU utilizza questi pin per controllare il resto del sistema e comunicargli quali operazioni intende compiere.

I pin di interrupt sono degli input che giungono alla CPU dalle periferiche. Nella maggior parte dei sistemi la CPU può comunicare a un dispositivo di I/O l'inizio di un'operazione e, mentre questo porta avanti il proprio lavoro, passa a eseguire qualche altra operazione. Non appena il dispositivo ha completato la propria operazione asserisce un segnale su uno di questi pin in modo da interrompere la CPU e permetterle di utilizzare il dispositivo, per esempio per controllare se si siano verificati degli errori di I/O. Alcune CPU hanno un pin di output per spedire una conferma in risposta a un segnale di interrupt.

I pin per l'arbitraggio del bus sono necessari per regolare il traffico sul bus, allo scopo di evitare che due dispositivi cerchino di usarlo nello stesso momento. Ai fini dell'arbitraggio la CPU conta quanto un altro dispositivo e anch'essa deve fare esplicita richiesta di utilizzo del bus.

Alcune CPU sono progettate per funzionare con coprocessori come i chip in virgola mobile oppure, in alcuni casi, i chip grafici o di altro tipo. Per facilitare la comunicazione tra CPU e coprocessore sono disponibili pin speciali per ricevere e soddisfare vari tipi di richieste.

Oltre a questi segnali alcune CPU possono avere altri pin per altri scopi, per esempio per fornire o ricevere informazioni di stato, o per resettare il calcolatore o ancora per garantire compatibilità con chip di I/O più vecchi.

3.4.2 Bus del calcolatore

Un **bus** è un collegamento elettrico che unisce diversi dispositivi. I bus possono essere classificati in base alla loro funzione; alcuni di loro sono impiegati internamente alla CPU per trasferire dati da e verso la ALU, mentre altri sono esterni alla CPU e servono a connetterla con la memoria o con altri dispositivi di I/O. Ciascun tipo di bus soddisfa certi requisiti e gode di proprietà specifiche. In questo paragrafo e nei successivi ci concentreremo sui bus

che connettono la CPU alla memoria e ai dispositivi di I/O, mentre nel prossimo capitolo esamineremo in modo più dettagliato i bus interni alla CPU.

I primi PC avevano un unico bus esterno chiamato anche **bus di sistema**. Esso era composto da 50 a 100 fili paralleli di rame che si inserivano nella scheda madre e i cui connettori erano distanziati a intervalli regolari per permettere l'inserimento di memorie e schede di I/O. Generalmente i personal computer moderni hanno invece un bus specifico tra la CPU e la memoria e (almeno) un altro bus per le periferiche. La Figura 3.35 mostra un sistema minimale, composto da un bus di memoria e un bus di I/O.

Come spesso avviene in letteratura i bus sono stati disegnati nella figura come frecce "larghe". Vi è una sottile differenza tra una linea spessa e una linea intersecata da un trattino diagonale vicino al quale si legge un numero (di bit). Quando tutti i bit sono dello stesso tipo, cioè sono, per esempio, bit d'indirizzo oppure bit di dati, si usa comunemente il trattino diagonale. Quando invece sono coinvolte linee d'indirizzo, di dati e di controllo si utilizza con più frequenza la freccia spessa.

Mentre i progettisti della CPU sono liberi di utilizzare all'interno del chip il tipo di bus che preferiscono, per i bus esterni bisogna invece definire delle precise regole di funzionamento, che devono essere rispettate dai dispositivi a loro collegati. La cosa consente che anche schede progettate da altri produttori possano collegarsi correttamente al bus. L'insieme di queste regole è detto **protocollo del bus**. Inoltre devono essere definite le specifiche meccaniche ed elettriche in modo che le schede prodotte da terze parti abbiano le dimensioni corrette e i loro connettori siano meccanicamente compatibili con quelli della scheda madre in termini di tensione, temporizzazione, e così via.

Nel mondo dei calcolatori esiste un gran numero di bus ampiamente utilizzati. Alcuni fra i più conosciuti, sia attuali sia d'importanza storica, sono i seguenti (tra parentesi sono indicati i calcolatori che li hanno adottati): l'Omnibus (PDP-8), l'Unibus (PDP-11), il Multibus (8086), il bus VME (apparecchiature per i laboratori di fisica), il bus PC IBM (PC/XT), il bus ISA (PC/AT), il bus EISA (80386), il Microchannel (PS/2), il Nubus (Macintosh), il bus PCI (molti PC), il bus SCSI (molti PC e workstation), l'Universal Serial Bus (PC moderni) e il FireWire (elettronica di consumo). Il mondo sarebbe probabilmente

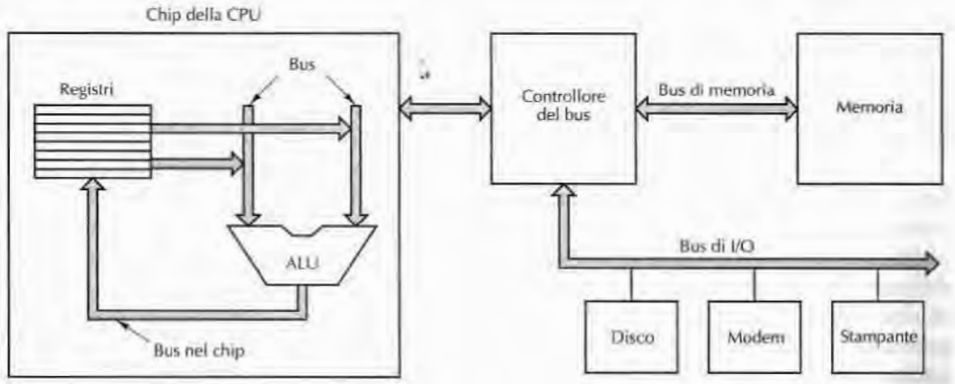


Figura 3.35 Sistema di un calcolatore con più bus.

Master	Slave	Esempio
CPU	Memoria	Prelievo delle istruzioni e dei dati
CPU	Dispositivo di I/O	Inizio del trasferimento dei dati
CPU	Coprocessore	Passaggio dell'istruzione al coprocessore da parte della CPU
I/O	Memoria	DMA (Direct Memory Access)
Coprocessore	CPU	Prelievo degli operandi dalla CPU da parte del coprocessore

Figura 3.36 Esempi di master e slave del bus.

migliore se tutti, tranne uno, sparissero di colpo dalla faccia della terra (d'accordo, è un po' troppo, ma che cosa dire se ne scomparissero tutti tranne due?). Sfortunatamente sembra molto difficile che avvenga una standardizzazione in quest'area, in quanto sono stati già fatti troppi investimenti in tutti questi sistemi, fra loro incompatibili.

Cominciamo il nostro studio partendo dal funzionamento dei bus. Alcune periferiche che si collegano al bus sono attive e possono iniziare un trasferimento dati, mentre altre sono passive e restano in attesa di una richiesta. Quelle attive sono chiamate ***master***, e quelle passive ***slave***. Quando la CPU ordina al controllore di un disco di leggere o scrivere un blocco, svolge il ruolo di master, e il controllore del disco quello di slave. Successivamente però il controllore del disco fungerà da master nel momento in cui ordina alla memoria di accettare le parole che sta leggendo dal disco. La Figura 3.36 elenca alcune tipiche combinazioni di master e slave. La memoria non può mai fungere da master.

Molto spesso i segnali digitali generati dalle periferiche sono troppo deboli per alimentare un bus, soprattutto se è relativamente lungo o se è collegato a molti dispositivi. Per questo motivo molti master sono connessi al bus mediante un chip chiamato **driver del bus**, che funge essenzialmente da amplificatore digitale; in modo analogo la maggior parte degli slave sono connessi al bus attraverso un **ricevitore del bus**. Per le periferiche che possono svolgere sia il ruolo di master sia quello di slave si utilizza un chip chiamato **trasmettitore-ricevitore del bus**. Spesso questi chip d'interfaccia sono dei dispositivi a tre stati per permettere loro di essere liberi (sconnessi) quando non sono necessari; un'altra possibilità, che permette di ottenere lo stesso effetto, consiste nell'agganciare la periferica al bus tramite un **collettore aperto**. Quando due o più dispositivi su una linea a collettore aperto asseriscono la linea nello stesso istante, il risultato è un OR di tutti i segnali; spesso questa organizzazione è chiamata OR-**cablata**. Nella maggior parte dei bus alcune linee sono a tre stati mentre altre, che richiedono la proprietà di essere OR-cablate, sono a collettore aperto.

Allo stesso modo della CPU anche il bus ha i propri indirizzi, i propri dati e le proprie linee di controllo. Ciononostante non è necessario che vi sia una corrispondenza uno-a-uno tra i pin della CPU e i segnali del bus. Alcune CPU hanno per esempio tre pin che codificano se si sta effettuando una lettura o una scrittura da memoria, una lettura o una scrittura di I/O oppure un'altra operazione. Un normale bus potrebbe avere una linea per la lettura da memoria, una seconda per la scrittura in memoria, una terza per la lettura da periferica, una quarta per la scrittura su periferica e così via. In questo caso si renderebbe necessario un decodificatore tra la CPU e il bus per far corrispondere le due estremità, cioè per convertire il segnale codificato in 3 bit in segnali distinti che possono essere spediti sulle varie linee del bus.

Il progetto e il funzionamento dei bus sono argomenti piuttosto complessi, sono trattati in vari libri (Anderson et al., 2004; Solari e Willse, 2004). Le principali decisioni da prendere nella progettazione di un bus riguardano l'ampiezza, la temporizzazione, l'arbitraggio e le operazioni che consente. Ciascuna di queste scelte ha un impatto significativo sulla velocità e sulla larghezza di banda del bus.

3.4.3 Ampiezza del bus

Nella progettazione dei bus il parametro più scontato da considerare è la sua ampiezza. Maggiore è il numero di linee d'indirizzo di un bus, maggiore sarà la quantità di memoria che la CPU potrà indirizzare direttamente. Se un bus ha n linee d'indirizzi, una CPU può indirizzare 2^n diverse locazioni di memoria. La cosa sembra piuttosto semplice.

Il problema è che bus più larghi richiedono un numero maggiore di fili rispetto a quelli più stretti. Questo significa anche una maggiore occupazione di spazio (per esempio sulla scheda madre) e la necessità di utilizzare connettori più grandi. Dato che tutti questi fattori rendono il bus più costoso occorre trovare un compromesso tra la dimensione massima di memoria e il costo del sistema. Un sistema con 64 linee d'indirizzi e 2^{32} byte di memoria costerà più di uno con 32 linee d'indirizzi e 2^{32} byte di memoria, ma nel secondo caso non è possibile espandere la memoria in un secondo momento.

Il risultato di questa osservazione è che molti progettisti di sistemi tendono a essere poco lungimiranti, con conseguenze negative. Il PC IBM originario conteneva una CPU 8088 e un bus con indirizzi a 20 bit, come mostrato nella Figura 3.37(a); con 20 bit a disposizione il PC poteva indirizzare soltanto 1 MB di memoria.

Con la CPU successiva, l'80286, Intel decise di aumentare lo spazio degli indirizzi a 16 MB; questo aumento richiese l'aggiunta di quattro linee di bus (senza modificare le 20 precedenti per ragioni di retrocompatibilità), come mostra la Figura 3.37(b). Purtroppo fu necessario aggiungere anche alcune nuove linee di controllo per gestire le nuove linee d'indirizzo. Quando apparve l'80386 furono aggiunte altre otto linee d'indirizzo oltre alle linee di controllo aggiuntive, come mostra la Figura 3.37(c). Il progetto risultante (il bus EISA)

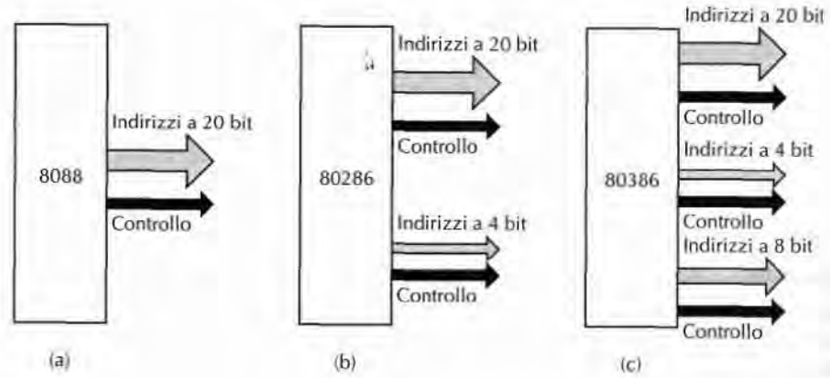


Figura 3.37 Crescita nel tempo degli indirizzi del bus.

è molto meno ordinato di come sarebbe potuto essere se si fosse partiti da zero avendo a disposizione fin dall'inizio tutte e 32 le linee.

Con il tempo, non solo tende a crescere il numero delle linee d'indirizzo, ma anche il numero delle linee di dati, seppur per una ragion diversa. Esistono due modi per aumentare la larghezza di banda dei dati su un bus: diminuire il periodo di clock del bus (più trasferimenti/s) oppure aumentare la larghezza dei dati del bus. Un'altra possibilità è quella di aumentare la velocità del bus, anche se ciò pone alcune difficoltà; i segnali su linee distinte viaggiano infatti a velocità leggermente diverse. Questo problema è conosciuto come **disallineamento del bus**, e più il bus è veloce più questo problema diventa marcato.

Un altro problema che sorge nel caso in cui si intenda velocizzare il bus è la perdita della retrocompatibilità. Schede progettate per bus più lenti non funzioneranno con il nuovo bus. Rendere inutilizzabili le vecchie schede non rende di certo felici né i proprietari di queste schede né i loro produttori. Per questa ragione l'approccio più utilizzato è quello di aggiungere nuove linee di dati, come mostrato nella Figura 3.37. In ogni caso, come ci si potrebbe aspettare, questa crescita incrementale non genera come risultato un'architettura semplice e ordinata. I PC IBM e i loro successori, per fare un esempio, sono passati da otto linee di dati a 16 e, in seguito, a 32 su un bus che è rimasto essenzialmente dello stesso tipo.

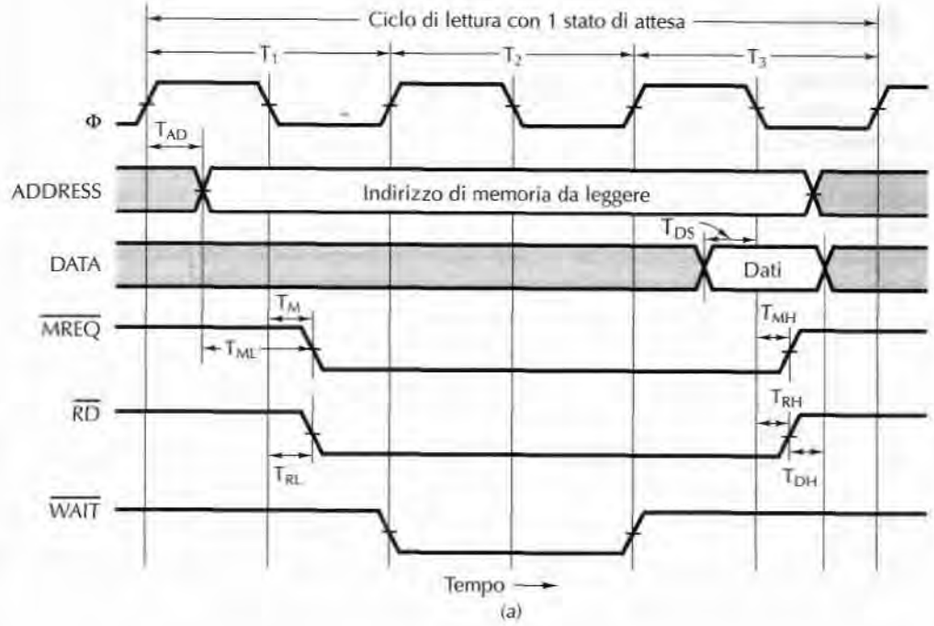
Per aggirare il problema di bus troppo ampi a volte i progettisti optano per un **bus multiplexato**. In questa architettura invece di tenere separate le linee d'indirizzo e quelle dei dati, si utilizza un certo numero di linee, per esempio 32, per entrambi. All'inizio di un'operazione sul bus le linee sono utilizzate per gli indirizzi, mentre in seguito vengono impiegate per i dati. Per esempio nel caso di una scrittura in memoria questo significa che le linee d'indirizzo devono essere impostate ai valori corretti e propagate fino alla memoria prima di spedire i dati sul bus. Quando si utilizzano linee separate è invece possibile spedire gli indirizzi e i dati allo stesso tempo. L'uso del multiplexing riduce l'ampiezza del bus (e quindi il costo), ma rende il sistema più lento. Quando devono prendere le proprie decisioni, i progettisti sono obbligati a valutare attentamente tutte queste possibilità.

3.4.4 Temporizzazione del bus

I bus possono essere separati in due categorie distinte in base alla loro temporizzazione. Un bus sincrono ha una linea pilotata da un oscillatore a cristalli: su questa linea un segnale consiste in un'onda quadra con frequenza generalmente compresa tra 5 e 100 MHz. Tutte le operazioni sul bus richiedono un numero intero di questi cicli, chiamati **cicli di bus**. L'altro tipo di bus, il bus asincrono, non ha invece un orologio principale; i cicli di bus possono avere una qualsiasi lunghezza e non devono essere necessariamente uguali nella comunicazione tra dispositivi.

Bus sincroni

Consideriamo la temporizzazione mostrata nella Figura 3.38(a) come esempio del funzionamento di un bus sincrono e riferiamoci a un clock a 100 MHz, che fornisce un ciclo di bus di 10 ns. Anche se questo valore potrebbe sembrare un po' lento rispetto alle velocità delle CPU, superiori a 3 GHz, occorre considerare che solo pochi bus dei PC esistenti sono molto più veloci. Per fare un esempio il bus PCI, molto diffuso, funziona generalmente a



Simbolo	Parametro	Min	Max	Unità
T_{AD}	Ritardo dell'output dell'indirizzo		4	nsec
T_{ML}	Indirizzo stabile prima di $\overline{MREQ}$	2		nsec
T_{M}	Ritardo di $\overline{MREQ}$ rispetto al fronte di discesa di Φ in T_1		3	nsec
T_{RL}	Ritardo di RD rispetto al fronte di discesa di Φ in T_1		3	nsec
T_{DS}	Tempo di impostaz. dei dati prima del fronte di discesa di Φ	2		nsec
T_{MH}	Ritardo di $\overline{MREQ}$ rispetto al fronte di discesa di Φ in T_3		3	nsec
T_{RH}	Ritardo di $\overline{RD}$ rispetto al fronte di discesa di Φ in T_3		3	nsec
T_{DH}	Tempo di mantenimento dei dati dopo la negazione di $\overline{RD}$	0		nsec

(b)

Figura 3.38 (a) Temporizzazione di una lettura su un bus sincrono. (b) Specifiche di alcuni tempi critici.

33 MHz oppure a 66 MHz. Le ragioni per le quali i bus attuali sono lenti sono già state sottolineate (disallineamento, retrocompatibilità, ecc).

Nel nostro esempio assumeremo inoltre che la lettura dalla memoria richieda 15 ns a partire dal momento in cui l'indirizzo è stabile. Con questi parametri, come vedremo a breve, saranno necessari tre cicli di bus per leggere una parola. Come mostra la figura, il primo ciclo inizia in corrispondenza del fronte di salita di T_1 e il terzo termina nel fronte

di salita di T_4. Si noti che i fronti di salita e di discesa non sono stati disegnati verticalmente in quanto nessun segnale elettrico può cambiare il proprio valore in un tempo nullo. In questo esempio assumeremo che il segnale impieghi 1 ns per cambiare valore. Il clock e le linee ADDRESS, DATA, $\overline{\texttt{MREQ}}$, $\overline{\texttt{RD}}$ e $\overline{\texttt{WAIT}}$ sono tutti mostrati usando la stessa scala temporale.

L'inizio di T_1 è definito dal fronte di salita del clock. A partire da T_1 la CPU fornisce l'indirizzo della parola sulle linee d'indirizzo. Dato che l'indirizzo non è un singolo valore, come invece il clock, non possiamo mostrarlo nella figura come una singola linea; viene quindi rappresentato con due linee che si incrociano nel momento in cui l'indirizzo cambia. L'ombreggiatura prima del punto d'intersezione indica inoltre che il valore non è importante. Adottando la stessa convenzione vediamo che il contenuto delle linee dei dati non è significativo fino a T_3.

Dopo che le linee d'indirizzo si sono stabilizzate sui nuovi valori, vengono asserite $\overline{\texttt{MREQ}}$ e $\overline{\texttt{RD}}$. La prima indica che si sta per accedere alla memoria (invece che a un dispositivo di I/O), mentre la seconda è asserita per le letture e negata per le scritture. Dato che la memoria impiega 15 ns dopo che l'indirizzo è stabile (inizia durante il primo ciclo di clock) non può fornire i dati richiesti durante T_2. La memoria asserisce la linea $\overline{\texttt{WAIT}}$ all'inizio di T_2 per segnalare alla CPU di non aspettarla.

Questa azione inserisce alcuni **stati di attesa** (cicli di bus addizionali) finché la memoria non completi l'operazione e neghi $\overline{\texttt{WAIT}}$. Nell'esempio è stato inserito uno stato di attesa (T_2), dato che la memoria è troppo lenta. All'inizio di T_3 la memoria nega $\overline{\texttt{WAIT}}$, in quanto è sicuro che i dati saranno disponibili nel corso del ciclo corrente.

Durante la prima metà di T_3, la memoria mette i dati sulle linee apposite. Nel fronte di discesa di T_3 la CPU consulta (cioè legge) le linee dei dati, memorizzando il valore in un suo registro. Dovendo leggere i dati la CPU nega $\overline{\texttt{MREQ}}$ e $\overline{\texttt{RD}}$. Se necessario può cominciare un altro ciclo di memoria nel successivo fronte di salita del clock. Questa sequenza può essere ripetuta indefinitamente.

Chiariamo ora il significato degli otto simboli presenti nel diagramma della Figura 3.38(b) che elenca le specifiche di temporizzazione. Per esempio, T_{AD} è l'intervallo di tempo tra il fronte di salita del ciclo di clock T_1 e il momento in cui vengono impostate le linee d'indirizzo. Secondo le specifiche di temporizzazione, $T_{AD} \leq 4$ ns. Questo significa che il produttore della CPU garantisce che durante ogni ciclo di lettura la CPU manderà in output l'indirizzo da leggere entro 11 ns dal punto medio del fronte di salita di T_1.

Le specifiche di temporizzazione richiedono inoltre che i dati siano disponibili sulle linee almeno T_{DS} ns (cioè 2 ns) prima del fronte di discesa di T_3. La cosa è necessaria per dare il tempo alla linea di stabilizzarsi prima che la CPU legga. La combinazione dei vincoli su T_{AD} e T_{DS} fa sì che, nel caso peggiore, la memoria abbia a disposizione soltanto $25 - 4 - 2 = 19$ ns tra il momento in cui appare l'indirizzo e quello in cui deve fornire i dati. Dato che sono sufficienti 10 ns, anche nel caso peggiore una memoria a 10 ns è sempre in grado di fornire la risposta durante il ciclo T_3. Al contrario una memoria a 20 ns potrebbe non farcela in tempo e in tal caso dovrebbe inserire un secondo stato di attesa e rispondere durante T_4.

Le specifiche di temporizzazione garantiscono inoltre che l'indirizzo venga impostato almeno 2 ns prima che $\overline{\texttt{MREQ}}$ sia asserito. Questo intervallo temporale può essere significativo nel caso in cui $\overline{\texttt{MREQ}}$ piloti la selezione del chip sulla memoria; questo poiché alcune memorie richiedono un certo tempo per l'impostazione dell'indirizzo prima di selezionare il chip. È chiaro che un progettista di CPU non dovrebbe scegliere un chip di memoria che richiede un tempo di setup di 3 ns.

I vincoli su T_M e T_{RL} fanno sì che $\overline{\texttt{MREQ}}$ e $\overline{\texttt{RD}}$ vengano asseriti entro 3 ns dalla discesa del clock T_1. Nel caso peggiore il chip di memoria, per mettere i propri dati sul bus, avrà a disposizione solo $10 + 10 - 3 - 2 = 15$ ns dopo che $\overline{\texttt{MREQ}}$ e $\overline{\texttt{RD}}$ sono stati asseriti. Questo vincolo si aggiunge (e ne è indipendente) all'attesa di 15 ns necessaria prima che l'indirizzo sia stabile.

T_{MH} e T_{RH} indicano quanto tempo impiegano $\overline{\texttt{MREQ}}$ e $\overline{\texttt{RD}}$ per essere negati dopo che i dati sono stati generati. Infine T_{DH} specifica per quanto tempo la memoria deve mantenere i dati sul bus dopo che $\overline{\texttt{RD}}$ è stato negato. Per quanto riguarda il nostro esempio di CPU, la memoria può rimuovere i dati dal bus non appena viene negato $\overline{\texttt{RD}}$; tuttavia su alcune CPU reali i dati devono essere mantenuti stabili per un tempo leggermente più lungo.

È importante sottolineare che la Figura 3.38 è una versione decisamente semplificata dei reali vincoli di temporizzazione. Anche se nella realtà vengono specificati molti altri tempi critici, l'esempio riesce tuttavia a dare l'idea del funzionamento di un bus sincrono.

Un altro punto che vale la pena precisare è che i segnali di controllo possono essere asseriti con valore alto o con valore basso. Tocca al progettista del bus determinare quale dei due sia più conveniente, e la scelta è essenzialmente arbitraria. Questa decisione potrebbe essere vista come l'equivalente, in hardware, della scelta che un programmatore compie quando deve decidere se rappresentare un settore vuoto del disco mediante valori 0 o valori 1.

Bus asincroni

Lavorare con i bus sincroni è facile per via dei loro intervalli temporali discreti; ciononostante presentano alcuni problemi. Per fare un esempio, qualsiasi operazione sul bus si svolge in tempi multipli del clock del bus; se una CPU e una memoria sono in grado di completare un trasferimento in 3,1 cicli, sono tuttavia obbligate ad allungare il tempo necessario a 4 cicli, dato che non sono consentite frazioni di cicli.

Inoltre, una volta scelto un ciclo di bus e appositamente costruite le memorie e le schede di I/O, è difficile trarre vantaggio dai successivi sviluppi della tecnologia. Supponiamo per esempio che alcuni anni dopo la realizzazione del sistema della Figura 3.38 si rendano disponibili nuove memorie con tempi di accesso di 8 ns invece che di 15 ns. Con la nuova memoria si eliminerebbe il tempo di attesa, accelerando di conseguenza la macchina. Supponiamo ora che diventino disponibili memorie a 4 ns; ciò non porterebbe ad alcun ulteriore guadagno in termini di prestazioni, poiché con questa architettura il tempo minimo per una lettura è di due cicli.

In altri termini, se un bus sincrono ha un insieme eterogeneo di dispositivi, alcuni veloci, altri più lenti, il bus deve essere regolato alla velocità della periferica più lenta, e le altre, pur essendo più veloci, non possono sfruttare tutto il loro potenziale.

È possibile gestire tecnologie caratterizzate da prestazioni diverse mediante l'utilizzo di un bus asincrono, cioè di un bus che non possiede un clock principale, come mostra la Figura 3.39. Il master del bus, invece di legare ogni operazione al clock, dopo aver asserito l'indirizzo, $\overline{\texttt{MREQ}}$, $\overline{\texttt{RD}}$ e tutto ciò che gli necessita, asserisce uno speciale segnale che chiameremo $\overline{\texttt{MSYN}}$ (*Master SYNchronization*). Quando lo slave vede questo segnale esegue il lavoro richiesto alla massima velocità possibile; una volta terminato il proprio compito asserisce $\overline{\texttt{SSYN}}$ (*Slave SYNchronization*).

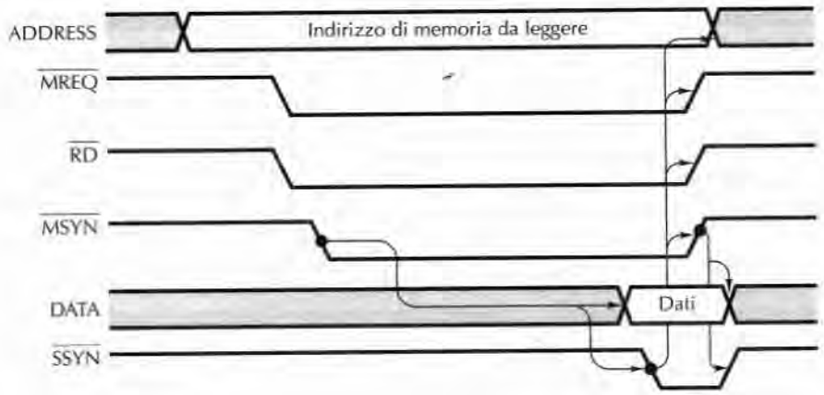


Figura 3.39 Funzionamento di un bus asincrono.

Quando il master vede che SSYN è stato asserito, sa che i dati sono disponibili e può quindi memorizzarli; successivamente nega le linee d'indirizzo, MREQ, RD e MSYN. Quando lo slave vede la negazione di MSYN, sa che il ciclo è stato completato e quindi nega SSYN; a questo punto si è tornati alla situazione iniziale, nella quale tutti i segnali sono negati e si attende il master successivo.

Come mostra la Figura 3.39 i diagrammi di temporizzazione dei bus asincroni (e talvolta anche dei bus sincroni) usano frecce per indicare cause ed effetti. L'asserzione di MSYN ha come effetto l'asserzione delle linee dei dati e il fatto che lo slave asserisce SSYN. L'asserzione di SSYN causa, a sua volta, l'asserzione delle linee d'indirizzo, di MREQ, di RD e di MSYN. La negazione di MSYN provoca infine la negazione di SSYN, che termina la lettura e riporta il sistema al suo stato originario.

Un insieme di segnali che coordina in questo modo due dispositivi con lo scopo di non farli interferire è chiamato **full handshake** ("stretta di mano completa"); esso consiste essenzialmente di quattro eventi:

1. MSYN è asserito
2. SSYN è asserito in risposta a MSYN
3. MSYN è negato in risposta a SSYN
4. SSYN è negato in risposta alla negazione di MSYN.

Dovrebbe essere chiaro che i full handshake non dipendono dalla temporizzazione. Ogni evento è causato da un evento precedente e non di un impulso del clock. Se una particolare coppia master-slave è lenta essa non influisce in alcun modo su una successiva coppia master-slave la cui velocità potrebbe essere molto più elevata.

Il vantaggio di un bus asincrono dovrebbe ora apparire evidente. Il problema è che in realtà la maggior parte dei bus è sincrona, in quanto più facile da realizzare; semplicemente la CPU asserisce i propri segnali e la memoria reagisce di conseguenza. Non è presente alcun feedback (causa e effetto), ma, nel caso in cui i componenti siano stati scelti

in modo appropriato, tutto funzionerà correttamente senza dover ricorrere all'handshake. Inoltre occorre dire che gli investimenti effettuati sulla tecnologia dei bus sincroni sono enormi.

3.4.5 Arbitraggio del bus

Finora abbiamo assunto tacitamente che la CPU sia l'unico master del bus. In realtà anche i chip di I/O devono diventare master per poter leggere e scrivere in memoria e provocare interrupt, e anche i coprocessori potrebbero avere la necessità di fungere da master. La domanda che sorge è: "Che cosa succede se due o più dispositivi vogliono diventare master del bus nello stesso momento?". La risposta è che si rende necessario qualche forma di **arbitraggio del bus** per evitare confusione.

Il meccanismo di arbitraggio può essere centralizzato o decentralizzato. Consideriamo inizialmente l'arbitraggio centralizzato, di cui una forma particolarmente semplice è mostrata nella Figura 3.40(a). In questo schema un singolo arbitro del bus determina chi sarà il prossimo master. In molte CPU l'arbitro è integrato nel chip stesso della CPU, mentre in altre è necessario utilizzare un chip separato. Il bus contiene un'unica linea di richiesta OR-cablata che in qualsiasi momento può essere asserita da uno o più dispositivi. L'arbitro non ha alcun modo per sapere quanti dispositivi hanno richiesto il bus; esso può solo distinguere tra "qualche richiesta" e "nessuna richiesta".

Quando l'arbitro vede una richiesta di utilizzo del bus, lo concede asserendo la linea per la concessione del bus. A questa linea sono collegati in serie tutti i dispositivi di I/O, come un semplice filo di lucine natalizie. Quando il dispositivo fisicamente più vicino all'arbitro vede la concessione, effettua un controllo per verificare se ne ha fatto richiesta. In caso affermativo si impossessa del bus, senza però propagare la concessione lungo il resto della linea. Se invece non ha fatto richiesta, propaga la concessione sulla linea in direzione del prossimo dispositivo che si comporterà allo stesso modo, e così pure i successivi, finché un dispositivo accetterà la concessione e si impossesserà del bus. Questo schema è chiamato **collegamento a festone** (*daisy chaining*) ed ha la proprietà di assegnare ai dispositivi diverse priorità in base alla loro vicinanza all'arbitro: il dispositivo più vicino vince.

Molti bus, per aggirare il fatto che le priorità sono implicitamente determinate dalla distanza rispetto all'arbitro, definiscono dei livelli di priorità. Per ciascun livello esiste una linea per effettuare la richiesta e una per segnalare la concessione. Il bus della Figura 3.40(b) ha due livelli, 1 e 2 (i bus reali possono averne 4, 8 o 16). A ciascun dispositivo viene assegnato un livello per la richiesta del bus; a quelli per cui la temporizzazione è più critica vengono assegnate priorità più elevate. Nella Figura 3.40(b) i dispositivi 1, 2 e 4 usano la priorità 1, mentre i dispositivi 3 e 5 usano la priorità 2.

Se in uno stesso momento ci sono richieste da livelli di priorità diversi l'arbitro concede il bus a un dispositivo di priorità più elevata; fra dispositivi della stessa priorità si utilizza invece la strategia del collegamento a festone. Nella Figura 3.40(b) in caso di conflitti il dispositivo 2 ha priorità sul dispositivo 4, che ce l'ha a sua volta sul 3. Il dispositivo 5 ha invece la priorità più bassa fra tutti, in quanto si trova alla fine del collegamento a festone a priorità inferiore.

Aprendo una parentesi occorre precisare che non è tecnicamente necessario collegare serialmente i dispositivi 1 e 2 alla linea di concessione del bus di livello 2, dato che non possono effettuare delle richieste su tale linea. Tuttavia per ragioni d'implementazione è più facile

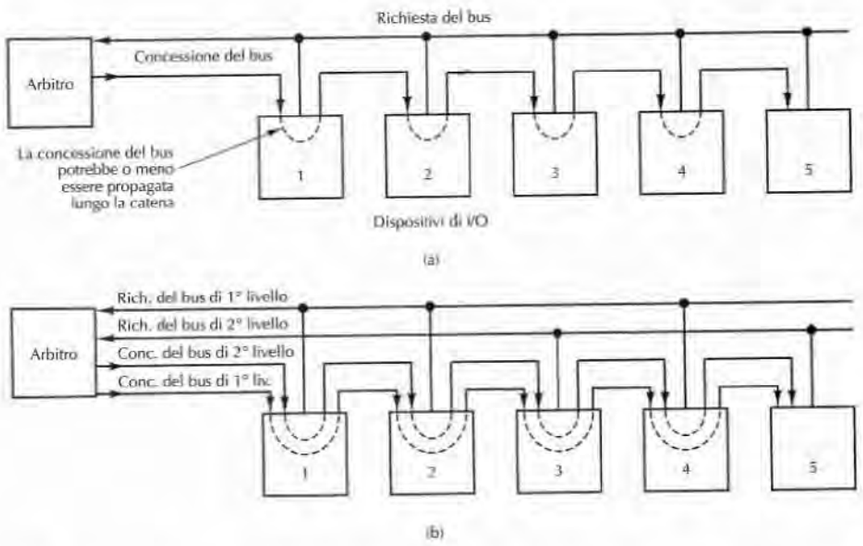


Figura 3.40 (a) Arbitro del bus centralizzato e a un livello che utilizza un collegamento a festone. (b) Stesso arbitro, ma a due livelli.

collegare tutte le linee di concessione a tutti i dispositivi, piuttosto che instaurare dei collegamenti speciali dipendenti dalla priorità associata a ciascun dispositivo.

Alcuni arbitri hanno una terza linea che viene asserita da un dispositivo nel momento in cui accetta una concessione e si impadronisce del bus. Subito dopo aver asserito questa linea di conferma, il dispositivo può negare le linee di richiesta e di concessione. Il risultato è che altri dispositivi possono richiedere il bus mentre il primo lo sta ancora utilizzando. Nel momento in cui termina il trasferimento in corso, il successivo master del bus sarà già stato selezionato. Esso può cominciare subito dopo che la linea di conferma è stata negata, cioè nel momento in cui può iniziare il nuovo turno determinato dall'arbitraggio. Questo schema richiede una linea aggiuntiva nel bus e una maggior quantità di componenti logici in ciascun dispositivo, ma permette un miglior utilizzo dei cicli del bus.

Nei sistemi in cui la memoria è collegata al bus principale, la CPU deve competere con tutti i dispositivi di I/O praticamente a ogni ciclo. Una soluzione molto comune per questa situazione consiste nell'assegnare alla CPU la priorità più bassa in modo che possa usare il bus soltanto quando nessun altro lo vuole. L'idea sottostante è che la CPU può sempre aspettare, mentre molto spesso i dispositivi di I/O sono obbligati ad acquisire il bus molto velocemente, pena la perdita dei dati in arrivo. I dischi che ruotano ad alte velocità, per esempio, non possono aspettare. In molti calcolatori questo problema viene risolto mettendo la memoria su un bus separato rispetto ai dispositivi di I/O, in modo che la CPU non debba competere con questi per l'accesso al bus.

Un'altra soluzione possibile è la decentralizzazione dell'arbitraggio del bus. Un calcolatore potrebbe per esempio avere 16 linee di richiesta, ciascuna con la propria priorità. Quando un dispositivo vuole utilizzare il bus asserisce la linea di richiesta. Tutti i dispositivi

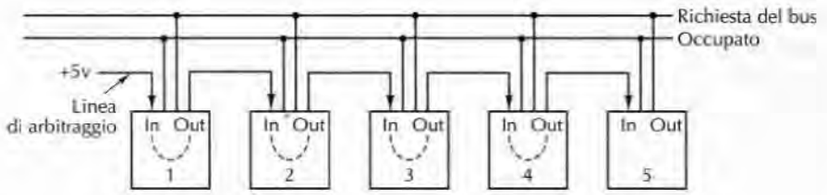


Figura 3.41 Arbitraggio decentralizzato del bus.

monitorano tutte le linee di richiesta in modo che alla fine di ciascun ciclo del bus ognuno di loro può sapere se era il richiedente con priorità più elevata e se quindi ha diritto a utilizzare il bus durante il ciclo successivo. Rispetto al metodo centralizzato questo schema di arbitraggio richiede un maggior numero di linee di bus, ma evita il potenziale costo dell'arbitro. Un altro limite è che il numero di dispositivi non può superare il numero delle linee di richiesta.

Un altro tipo di arbitraggio decentralizzato del bus, mostrato nella Figura 3.41, utilizza solamente tre linee, indipendentemente da quanti dispositivi sono presenti. La prima linea del bus è una linea OR-cablata per effettuare le richieste al bus. La seconda linea del bus è chiamata `BUSY` ed è asserita dal master corrente. La terza è invece utilizzata per arbitrare il bus ed è un collegamento a festone fra tutti i dispositivi; la testa di questa catena viene mantenuta asserita collegandola a un'alimentazione di 5 volt.

Quando nessun dispositivo richiede il bus la linea di arbitraggio, che è asserita, viene propagata attraverso tutti i dispositivi. Per acquisire il bus un dispositivo deve prima controllare se il bus è inattivo e se il segnale dell'arbitraggio che sta ricevendo, `IN`, è asserito. Se `IN` è negato, non può diventare il master e nega `OUT`. Se invece `IN` è asserito, il dispositivo nega `OUT` in modo che il dispositivo successivo nel collegamento a festone veda `IN` negato e neghi anch'esso il proprio `OUT`. Tutti i dispositivi successivi vedranno pertanto `IN` negato e negheranno a loro volta `OUT`. Una volta tornata la calma solo un dispositivo rimarrà con `IN` asserito e `OUT` negato; esso diventerà il master del bus, asserirà `BUSY` e `OUT` e inizierà il proprio trasferimento.

Con un semplice ragionamento si deduce che fra i dispositivi che richiedono il bus lo ottiene quello che si trova più a sinistra. Questo schema è quindi simile all'originario arbitraggio con collegamento a festone, tranne per il fatto che non c'è l'arbitro; questa differenza lo rende più economico, più veloce e non soggetto ai potenziali guasti dell'arbitro.

3.4.6 Operazioni del bus

Fino a questo momento abbiamo trattato solamente bus ordinari con un master (generalmente la CPU) che legge da uno slave (di solito la memoria) o scrive su di esso. Nei casi reali sono possibili anche altri tipi di bus che ora analizzeremo.

Di solito viene trasferita una parola alla volta. Tuttavia quando si utilizza la cache è preferibile prelevare in una volta sola un'intera linea di cache (per esempio 16 parole consecutive a 32 bit). Spesso i trasferimenti di blocchi possono essere effettuati in modo più efficiente rispetto a una sequenza di trasferimenti individuali. All'inizio della lettura di un blocco il master del bus comunica allo slave quante parole devono essere trasferite inserendo, per esempio

durante T_1, il conteggio delle parole sulle linee dei dati. Lo slave, invece di restituire un'unica parola, genera in output a ogni ciclo una nuova parola finché il contatore non si sia esaurito. La Figura 3.42 mostra una versione modificata della Figura 3.38(a) in cui un segnale aggiuntivo $\overline{\texttt{BLOCK}}$ viene asserito per indicare che è stato richiesto il trasferimento di un blocco. In questo esempio una lettura di un blocco di 4 parole richiede 6 cicli invece che 12.

Esistono anche altri tipi di cicli di bus. Per esempio su un sistema multiprocessore con due o più CPU sullo stesso bus è spesso necessario assicurarsi che, in un dato momento, soltanto una CPU utilizzi delle strutture dati critiche della memoria. Una tipica soluzione del problema consiste nell'avere in memoria una variabile che vale 0 quando nessuna CPU sta utilizzando la struttura dati e 1 quando invece è in uso. Se una CPU vuole acquisire l'accesso alla struttura dati deve leggere il valore della variabile e impostarla a 1, se vale 0. Il problema è che, con una buona dose di sfortuna, due CPU potrebbero leggere la variabile durante cicli di bus consecutivi. Se entrambe vedono che la variabile è 0 allora entrambe la imposteranno a 1 e penseranno di essere l'unica CPU che sta utilizzando la struttura dati. Questa sequenza di eventi porta al caos.

Per evitare questa situazione i sistemi multiprocessore hanno spesso uno speciale ciclo di bus leggi-modifica-scrivi che consente a una qualsiasi CPU di leggere una parola dalla memoria, analizzarla e riscriverla in memoria, tutto senza rilasciare il bus. Questo tipo di ciclo evita che altre CPU che intendono utilizzare il bus riescano a impadronirsene interferendo con l'operazione della prima CPU.

Un altro importante tipo di ciclo di bus serve a gestire gli interrupt. Quando la CPU ordina a un dispositivo di I/O di effettuare qualche operazione si aspetta di solito un interrupt alla fine del lavoro; la segnalazione di questi interrupt richiede l'utilizzo del bus.

Dato che più dispositivi potrebbero voler generare un interrupt simultaneamente, si verificano gli stessi tipi di problemi di arbitraggio visti nel caso dei cicli di bus ordinari. La soluzione

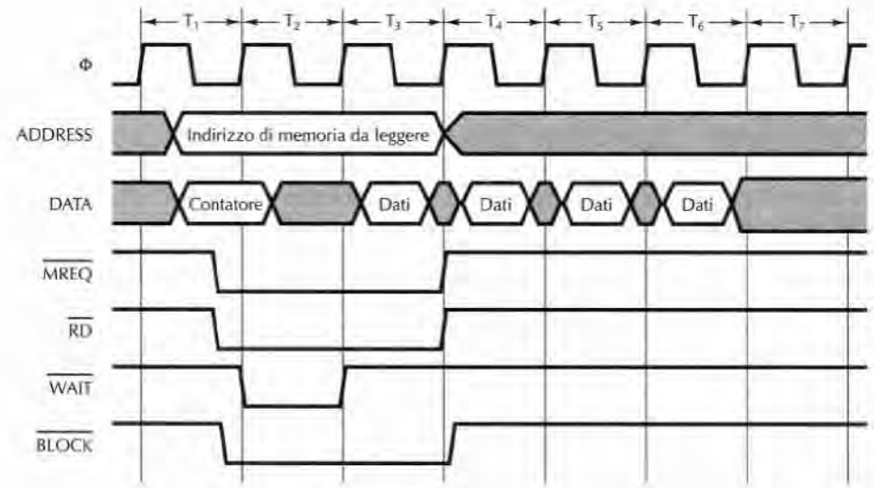


Figura 3.42 Trasferimento di un blocco.

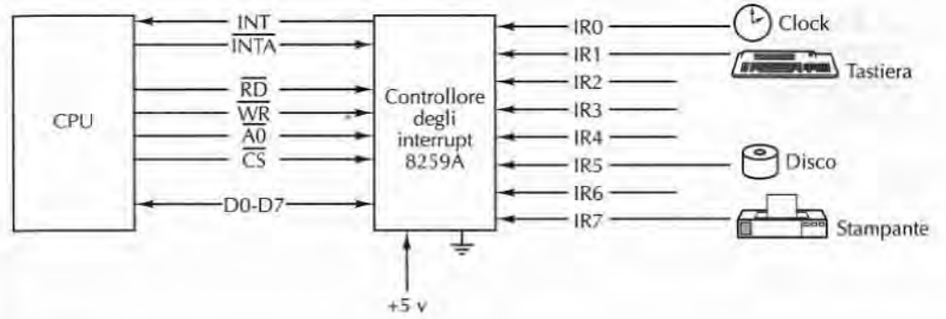


Figura 3.43 Utilizzo del controllore degli interrupt 8259A.

più usuale prevede di assegnare priorità ai dispositivi e di utilizzare un arbitro centralizzato per dare priorità al dispositivo per cui la temporizzazione è più critica. Esistono chip controllori di interrupt di tipo standard che sono utilizzati in modo molto diffuso; il PC IBM e tutti i suoi successori utilizzano il chip Intel 8259A, illustrato nella Figura 3.43.

Il chip 8259A ha otto input, chiamati IRx (*Interrupt Request*, "richiesta di interrupt"), con i quali si possono collegare direttamente fino a otto controllori di I/O. Quando uno di questi dispositivi vuole causare un interrupt asserisce la propria linea di input. Quando uno o più input sono asseriti il chip 8259A asserisce INT (*INTerrupt*), che pilota direttamente il pin di interrupt sulla CPU. Quando la CPU è in grado di gestire l'interrupt rispedisce all'8259A un impulso su INTA (*INTerrupt Acknowledge*, "conferma dell'interrupt"); a quel punto l'8259A deve specificare quale input ha causato l'interrupt generando in output il suo numero sul bus dei dati. Questa operazione richiede un ciclo di bus speciale. L'hardware della CPU utilizza quindi quel numero come indice all'interno di una tabella di puntatori, chiamata **vettore di interrupt**, per cercare l'indirizzo della procedura da eseguire per poter gestire l'interrupt.

L'8259A ha al suo interno vari registri che la CPU può leggere e scrivere utilizzando i cicli di bus ordinari e i pin $\overline{\texttt{RD}}$ (*ReaD*), $\overline{\texttt{WR}}$ (*WRite*), $\overline{\texttt{CS}}$ (*Chip Select*) e $\overline{\texttt{A0}}$. Quando il software ha gestito l'interrupt ed è pronto per accettarne uno nuovo scrive uno speciale codice all'interno di uno dei registri; ciò fa sì che l'8259A neghi INT, a meno che non vi sia un altro interrupt pendente. In questi registri si può anche scrivere per portare l'8259A in uno dei vari modi disponibili, mascherando per esempio un insieme di interrupt oppure abilitando altre funzionalità.

Quando sono presenti più di otto dispositivi di I/O è possibile utilizzare in cascata più chip 8259A. Nel caso più estremo tutti gli otto input possono essere connessi agli output di ulteriori otto 8259A; è possibile così gestire fino a 64 dispositivi di I/O in una rete di interrupt a due livelli. L'8259A possiede alcuni pin dedicati alla gestione di questa cascata, ma qui ci è sembrato opportuno ometterli.

Non abbiamo affatto esaurito il tema della progettazione dei bus; tuttavia il materiale presentato finora dovrebbe fornire sufficienti conoscenze di base per capire qual è in sostanza il funzionamento di un bus e come le CPU e i bus possono interagire. Ci spostiamo ora dal generale allo specifico e diamo uno sguardo ad alcuni esempi di CPU commerciali e ai loro bus.

3.5 Esempi di chip della CPU

In questo paragrafo esamineremo in modo dettagliato tre chip: il Pentium 4, l'UltraSPARC III e l'8051.

3.5.1 Pentium 4

Il Pentium 4 è un diretto discendente della CPU 8088 utilizzata nel PC IBM originario. Il primo Pentium 4 fu introdotto nel novembre del 2000 sotto forma di una CPU funzionante a 1,5 GHz con 42 milioni di transistor e con una *larghezza di linea* da 0,18 micron. La larghezza di linea corrisponde alla grandezza dei collegamenti tra i transistor (ed è anche una misura della dimensione dei transistor stessi). Più stretta è la larghezza di linea e più transistor possono essere inseriti sul chip. La legge di Moore riguarda fondamentalmente la capacità di continuare a ridurre le larghezze di linea. Larghezze di linea più piccole permettono velocità di clock più elevate. Per fare un confronto con le misure in gioco si pensi che i capelli hanno un diametro che varia tra 20 micron e 100 micron, di cui i più fini sono quelli biondi, mentre quelli con diametro maggiore sono quelli neri.

Nel corso degli ultimi tre anni Intel, acquisendo esperienza con il processo produttivo, è evoluta fino a raggiungere 55 milioni di transistor funzionanti alla velocità di 3,2 GHz e con una larghezza di linea di 0,09 micron. Il Pentium 4, pur essendo decisamente più avanzato rispetto all'8088 con 29.000 transistor, è completamente compatibile con esso e può eseguire i suoi programmi senza alcuna modifica (lo stesso vale, ovviamente, anche per i programmi di tutti i processori intermedi).

Dal punto di vista del software il Pentium 4 è una macchina interamente a 32 bit. A livello utente ha tutte le funzionalità ISA dei modelli 80386, 80486, Pentium, Pentium II, Pentium Pro e Pentium III, compresi gli stessi registri, le stesse istruzioni e la stessa implementazione dello standard in virgola mobile IEEE 754 interamente sul chip, inoltre ha alcune nuove istruzioni dedicate alle applicazioni multimediali.

Dal punto di vista hardware il Pentium 4 è tuttavia una macchina parzialmente a 64 bit, dato che può scambiare dati con la memoria in unità di 64 bit. Questi trasferimenti a 64 bit, anche se non visibili al programmatore, rendono la macchina più veloce di quanto lo sarebbe una macchina puramente a 32 bit.

Internamente, a livello di microarchitettura, il Pentium 4 è radicalmente differente rispetto a tutti i suoi predecessori. Quelli immediatamente precedenti, cioè i Pentium II, Pro, e III, usavano la stessa microarchitettura interna (chiamata P6), e differivano soltanto per la velocità e per altri dettagli di minor importanza. Il Pentium 4 adotta invece una nuova microarchitettura (chiamamta NetBurst) che si differenzia in modo sostanziale da P6. Essa è dotata di una pipeline più profonda, due ALU (funzionanti a doppia frequenza di clock per permettere due operazioni per ciclo) e supporta l'*hyperthreading*. Quest'ultima caratteristica fornisce due insiemi di registri e alcune altre risorse interne per permettere al Pentium 4 di alternare l'esecuzione di due programmi molto velocemente, come se il calcolatore contenesse fisicamente due CPU. La microarchitettura verrà analizzata nel corso del Capitolo 4. Anche il Pentium 4, come i suoi predecessori, è tuttavia in grado di svolgere l'esecuzione di più istruzioni nello stesso tempo; ciò rende il processore una macchina superscalare.

Alcuni modelli del Pentium 4 hanno una cache a due livelli, mentre altri ne hanno tre. Tutti i modelli hanno sul chip una cache SRAM di primo livello (L1) di 8 KB. Diversamente dalla cache L1 del Pentium III, che memorizzava semplicemente i byte grezzi di memoria, quella del Pentium 4 si spinge oltre. Quando le istruzioni vengono prelevate dalla memoria, sono convertite in micro-operazioni per essere effettivamente eseguite dal cuore RISC del Pentium 4. La cache L1 sul Pentium 4 può contenere fino a 12.000 micro-operazioni già decodificate, evitando così di doverle decodificare più volte. Nei modelli più vecchi la cache di secondo livello gestisce fino a 256 KB di memoria, mentre in quelli più recenti la quantità di memoria arriva fino a 1 MB. Nella cache L2 si memorizzano i byte dalla memoria tali e quali, senza alcuna decodifica, e al suo interno è possibile conservare sia codice sia dati. Il Pentium 4 Extreme Edition, per spingere ancora più in avanti le prestazioni, ha un'ulteriore cache di terzo livello di 2 MB.

Dato che tutti i chip del Pentium 4 hanno almeno due livelli di cache, nei sistemi multiprocessore sorge un problema nel momento in cui una CPU modifica una parola nella propria cache. Dato che le parole modificate all'interno della cache non vengono riscritte immediatamente in memoria, se un'altra CPU prova a leggere quella parola dalla memoria otterrà un valore non aggiornato. Per garantire la coerenza della memoria in un multiprocessore ciascuna CPU effettua uno **snooping** ("ficcanaso") sul bus della memoria per cercare eventuali riferimenti alle parole che ha memorizzato nella propria cache. Quando vede un tale riferimento si intromette nel bus fornendo il dato richiesto prima che lo possa fare la memoria. Nel Capitolo 8 studieremo lo snooping.

Nei sistemi Pentium 4 si utilizzano due principali bus esterni, entrambi sincroni. Il bus della memoria è utilizzato per accedere alla (S)DRAM centrale, mentre il bus PCI è utilizzato per comunicare con i dispositivi di I/O. Talvolta un *legacy bus* (cioè un bus di antico retaggio, ossia vecchio) è collegato al bus PCI per permettere la connessione di periferiche più vecchie.

Una differenza sostanziale tra il Pentium 4 e tutti i suoi predecessori è il suo confezionamento. Un problema dei chip moderni riguarda la quantità di potenza che consumano e il calore che producono. Il Pentium 4, per esempio, consuma tra i 63 watt e gli 82 watt a seconda della frequenza. Per questo motivo Intel è continuamente alla ricerca di soluzioni per gestire il calore prodotto dai chip della CPU. Il Pentium 4 è contenuto in un involucro quadrato di 35 mm per lato, contenente 478 pin sul lato inferiore, 85 dei quali sono per l'alimentazione e 180 sono collegati alla terra per ridurre il disturbo. I pin sono disposti in una griglia quadrata 26 × 26, tranne che in un'area centrale grande 14 × 14. In uno dei lati del quadrato mancano inoltre due pin per evitare che il chip possa essere inserito in modo non corretto. La Figura 3.44 mostra la disposizione fisica dei pin.

Il chip è attrezzato in modo che sopra di esso sia possibile montare un dissipatore di calore e una ventola per raffreddarlo. Per avere un'idea dell'entità del problema basta accendere una lampadina da 60 Watt, aspettare qualche minuto e sfiorarla (senza però toccarla); questa quantità di calore deve essere dissipata continuamente. Di conseguenza, quando il Pentium 4 smette di essere utile come CPU, lo si può sempre riciclare come fornello da campeggio!

Secondo le leggi della fisica ogni fenomeno che produce una gran quantità di calore deve consumare molta energia. In un calcolatore portatile non è desiderabile utilizzare una gran quantità di energia in quanto la carica della batteria è limitata e tenderebbe a consu-

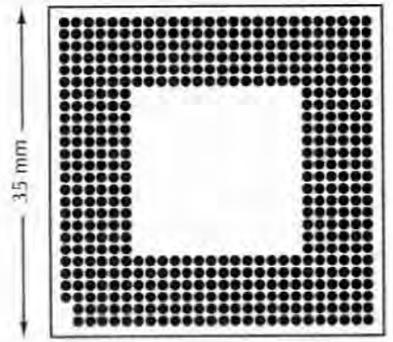


Figura 3.44 Disposizione fisica dei contatti del Pentium 4.

marsi molto rapidamente. Per risolvere questo problema Intel ha dotato le CPU di un modo per "riposare" quando sono inattive e di entrare in un "sonno profondo" quando è probabile che resteranno inattive per un certo periodo di tempo. Ci sono cinque stati possibili, che variano dalla piena esecuzione al sonno profondo. Negli stati intermedi alcune funzionalità (come lo snooping della cache e la gestione degli interrupt) sono disabilitate, mentre altre continuano a essere utilizzabili. Quando la CPU si trova nello stato di sonno profondo la cache e i registri sono preservati, ma il clock e tutte le unità interne sono disabilitate e, per risvegliarle, è necessario un segnale hardware. Non si sa se il Pentium 4 sia in grado di sognare quando dorme profondamente.

Disposizione logica dei contatti del Pentium 4

Fra i 478 pin del Pentium 4, 198 sono utilizzati per i segnali, 85 per l'alimentazione (a diverse tensioni), 180 per la terra e 15 per utilizzi futuri. Dato che alcuni segnali logici utilizzano due o più pin (come gli indirizzi di memoria richiesti), il numero di segnali distinti è solamente 56. La Figura 3.45 mostra una versione piuttosto semplificata della disposizione logica dei contatti. Sul lato sinistro della figura ci sono cinque gruppi principali di segnali per il bus della memoria. I nomi riportati interamente in maiuscolo sono gli effettivi "nomi Intel" dei segnali, mentre gli altri sono nomi collettivi per quei tipi di segnali.

È importante capire qual è la convenzione adottata da Intel per attribuire un nome ai segnali. Dato che al giorno d'oggi tutti i chip sono progettati al calcolatore occorre rappresentare i nomi dei segnali in ASCII. L'utilizzo di una lineetta sopra il nome per indicare che un segnale è asserito con valore basso sarebbe troppo complicato. Per questo motivo Intel usa un'altra convenzione, aggiungendo il simbolo # dopo il nome. $\overline{\texttt{BPRI}}$ viene quindi indicato come `BPRI#`. Come si può vedere nella figura la maggior parte dei segnali del Pentium 4 è asserita con valore basso.

Esaminiamo i segnali a partire da quelli del bus. Il primo gruppo è utilizzato per richiedere l'utilizzo del bus (esegue cioè l'arbitraggio del bus). `BR0` è utilizzato per richiedere il bus, mentre `BPRI#` permette a un dispositivo di effettuare una richiesta ad alta priorità, che ha la precedenza su quella ordinaria. `LOCK#` dà alla CPU la possibilità di bloccare il bus per evitare che altri dispositivi se ne impossessino prima del necessario.

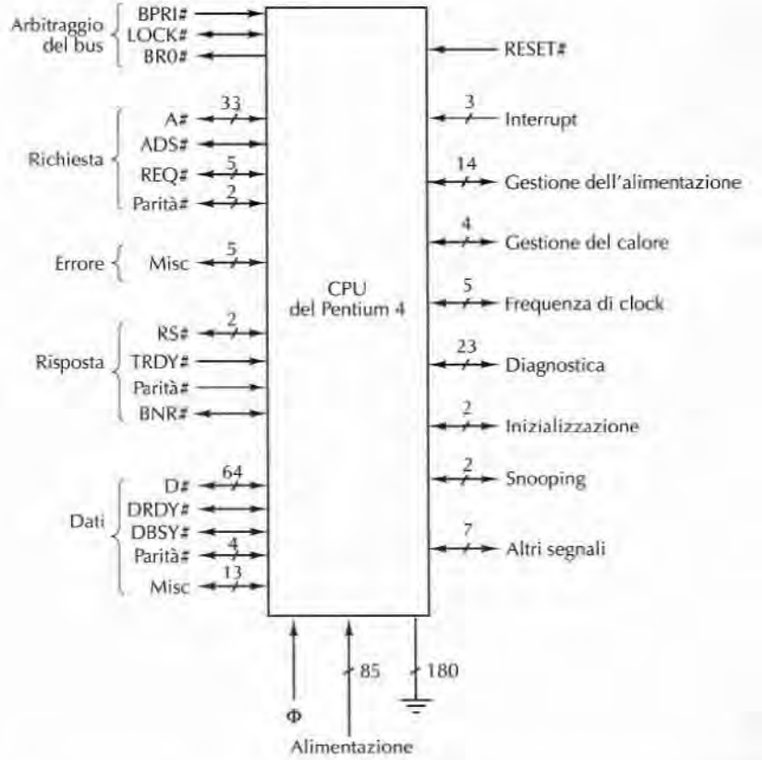


Figura 3.45 Disposizione logica dei contatti del Pentium 4. I nomi interamente in maiuscolo sono i nomi ufficiali di Intel per i singoli segnali; gli altri sono nomi di gruppi di segnali o descrizioni dei segnali.

Dopo aver acquisito il bus, una CPU o un altro master può effettuare una richiesta sul bus utilizzando il successivo gruppo di segnali. Gli indirizzi hanno 36 bit, ma i 3 meno significativi devono valere sempre 0; non hanno bisogno di alcun contatto e quindi `A#` ha soltanto 33 pin. Tutti i trasferimenti sono a 8 byte e sono allineati ai limiti di un gruppo di 8 byte. Con indirizzi a 36 bit la massima dimensione indirizzabile della memoria è di 2^{36} byte, pari a 64 GB.

Quando si scrive un indirizzo sul bus il segnale `ADS#` viene asserito per comunicare alla destinazione (per esempio la memoria) che le linee d'indirizzo sono valide. Successivamente si utilizzano le linee `REQ#` per spedire il tipo di ciclo di bus richiesto (per esempio leggere una parola oppure scrivere un blocco). Le due linee di parità proteggono `A#` e `REQ#`.

Le cinque linee degli errori sono utilizzate per riportare errori in virgola mobile, interni, sul controllo (cioè sull'hardware) della macchina o di altro tipo.

Il gruppo Risposta contiene i segnali utilizzati dallo slave per restituire un risultato al master. `RS#` contiene il codice di stato (*status code*) mentre `TRDY#` indica che lo slave è pronto

per accettare dati dal master. Anche su questi segnali viene effettuato un controllo di parità. BNR è utilizzato per asserire uno stato di attesa quando il dispositivo indirizzato non è in grado di rispondere in tempo.

L'ultimo gruppo è utilizzato per il trasferimento effettivo dei dati. D# è usato per inserire 8 byte di dati nel bus e la loro presenza viene annunciata asserendo DRDY#. DBSY# viene invece utilizzato per comunicare a tutti i dispositivi che in quel momento il bus è occupato. I segnali di dati miscellanei riguardano altri aspetti.

RESET# serve per reinizializzare la CPU in caso di calamità o quando l'utente preme il pulsante di reset sulla parte frontale del PC.

È possibile configurare il Pentium 4 in modo che utilizzi gli interrupt allo stesso modo dell'8088 (per la retrocompatibilità) oppure secondo una nuova modalità che impiega un dispositivo chiamato **APIC** (*Advanced Programmable Interrupt Controller*).

Il Pentium 4 può funzionare a varie tensioni, ma deve sapere quale di queste viene utilizzata. I segnali per la gestione dell'alimentazione sono utilizzati per la selezione automatica della tensione, per comunicare alla CPU che la tensione è stabile e per altri aspetti riguardanti l'alimentazione. Sempre mediante questi segnali vengono gestiti i vari stati di riposo, dato che la loro scelta è fatta allo scopo di ridurre il consumo energetico.

Il Pentium 4, nonostante una gestione sofisticata dell'alimentazione, si riscalda notevolmente. Il gruppo per la gestione del calore ha a che fare con la temperatura e permette alla CPU di comunicare all'ambiente circostante che si trova in pericolo di surriscaldamento. Se la sua temperatura interna raggiunge 130°C (266°F) viene asserito uno dei pin. Quando una CPU raggiunge questa temperatura significa che sta sognando di andare in pensione e progetta di diventare un fornello da campeggio!

Il gruppo di segnali della frequenza di clock deve determinare la frequenza del bus di sistema. Il gruppo per la diagnostica contiene invece segnali per sistemi di test e debugging secondo lo standard IEEE 1149.1 dei test JTAG. Il gruppo di inizializzazione gestisce il *boot*, cioè l'avvio del sistema.

L'ultimo gruppo è un calderone di segnali tra i quali ne troviamo uno che indica se la socket della CPU è occupata, uno che abilita l'emulazione 8088 e altri dedicati a utilizzi particolari.

Pipeline sul bus di memoria del Pentium 4

Le odierne CPU come il Pentium 4 sono molto più veloci rispetto alle moderne memorie DRAM. Per evitare che la CPU rimanga in attesa a causa della mancanza di dati è fondamentale ottenere dalla memoria il massimo throughput possibile. Per questa ragione il bus di memoria del Pentium 4 è altamente strutturato a pipeline, al punto che su di esso possono avvenire fino a otto transazioni contemporanee. Il concetto di pipeline è stato visto nel contesto delle CPU (Figura 2.4) nel corso del Capitolo 2, e anche le memorie possono avere un'architettura a pipeline.

Per consentire l'uso della pipeline, le richieste di memoria del Pentium 4, chiamate **transazioni**, sono composte da sei stadi.

1. Fase di arbitraggio del bus.
2. Fase di richiesta.

3. Fase di segnalazione dell'errore.
4. Fase di investigazione (*snoop*).
5. Fase di risposta.
6. Fase dei dati.

Non tutte le fasi sono necessarie per tutte le transazioni. La prima fase determina quale dei potenziali master del bus diventerà effettivamente tale. La seconda permette di inserire gli indirizzi nel bus e di effettuare richieste. La fase di segnalazione dell'errore consente allo slave di annunciare che l'indirizzo ha un errore di parità o che qualcosa è andato storto. La fase di investigazione (*snoop*), necessaria solo nei sistemi multiprocessore, permette a una CPU di indagare su un'altra. La fase di risposta è il momento in cui il master viene a sapere se sta per ricevere oppure no i dati che desiderava. Infine l'ultima permette il trasferimento effettivo dei dati verso la CPU che li ha richiesti.

Il segreto dell'organizzazione a pipeline del bus di memoria del Pentium 4 è che ciascuna fase utilizza diversi segnali del bus, essendo così completamente indipendente dalle altre. I sei gruppi di segnali richiesti sono quelli mostrati sulla sinistra della Figura 3.45. Per fare un esempio, una CPU può cercare di ottenere il bus utilizzando i segnali di arbitraggio. Dopo aver acquisito il diritto a procedere, rilascia queste linee e inizia a utilizzare il gruppo di linee relative a Richiesta. Nello stesso tempo un'altra CPU o un dispositivo di I/O può entrare nella fase di arbitraggio, e così via. La Figura 3.46 mostra come possono avvenire nello stesso momento più transazioni del bus.

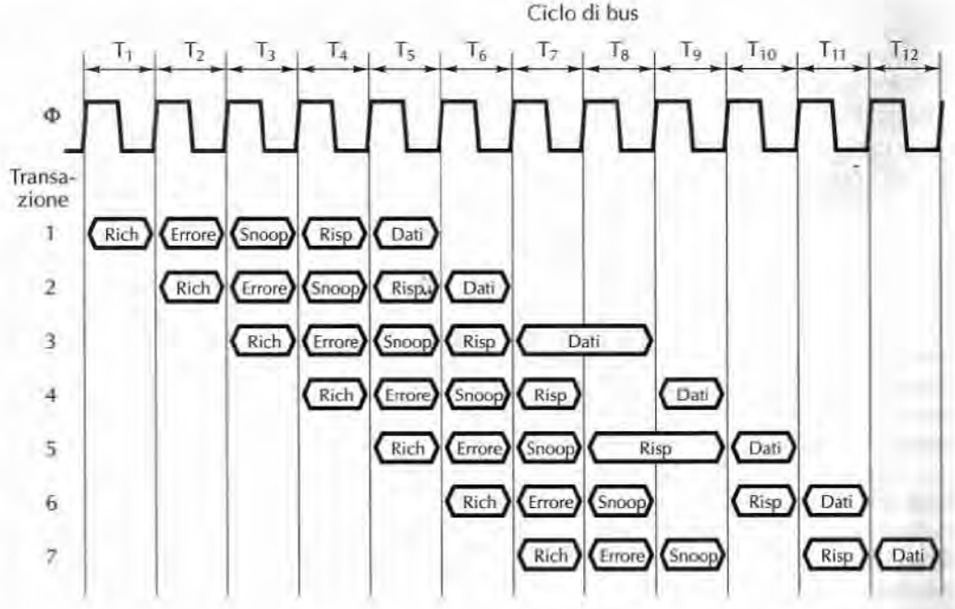


Figura 3.46 Richieste organizzate a pipeline sul bus di memoria del Pentium 4.

Nella Figura 3.46 non è mostrata la fase di arbitraggio, dato che non è sempre necessaria. Se, per esempio, il possessore corrente del bus (che spesso è la CPU) vuole eseguire un'altra transazione non deve riacquisire il bus, ma può semplicemente richiedere di continuare a utilizzarlo e concederlo poi a uno degli altri dispositivi che lo richiede. Le transazioni 1 e 2 sono dirette: cinque fasi in cinque cicli di bus. La transazione 3 introduce una fase di dati più lunga; ciò potrebbe essere dovuto al trasferimento di un blocco oppure per via del fatto che la memoria indirizzata ha inserito uno stato di attesa. Come conseguenza la transazione 4 non può iniziare la propria fase di dati quando vuole, ma deve osservare il segnale DBSY# e aspettare fintanto che non viene negato. Nella transazione 5 vediamo che anche la fase di risposta può richiedere più cicli di bus, ritardando quindi la transazione 6. Infine nella transazione 7 notiamo che una "bolla" inserita nella pipeline vi rimane se nei passi immediatamente successivi iniziano delle nuove transazioni. Nei casi reali le bolle non rimangono tuttavia a lungo, dato che è poco probabile che la CPU cerchi di iniziare nuove transazioni in corrispondenza di ogni ciclo di bus.

3.5.2 UltraSPARC III

Come secondo esempio di chip di CPU esamineremo ora la famiglia Sun UltraSPARC, composta da una linea di processori a 64 bit. Queste CPU seguono fedelmente l'architettura SPARC Version 9, che comprende anche le CPU a 64 bit, e sono impiegate nelle workstation e nei server Sun oltre che in diverse altre applicazioni. Questa famiglia comprende l'UltraSPARC I, II e III, che si differenziano principalmente per la data di immissione sul mercato, la velocità di clock e alcune istruzioni aggiuntive introdotte a ogni modello. Per essere più precisi, in seguito ci riferiremo all'UltraSPARC III, anche se la maggior parte delle informazioni sull'architettura (cioè quelle indipendenti dalla tecnologia usata) vale anche per gli altri modelli UltraSPARC.

L'UltraSPARC III è una tradizionale macchina RISC ed è pienamente compatibile a livello binario con l'architettura SPARC Version 8 a 32 bit; pertanto è in grado di eseguire i programmi macchina della SPARC V8 senza alcuna modifica. L'unico aspetto nel quale l'UltraSPARC III si distingue dall'architettura SPARC V9 è nell'aggiunta delle istruzioni VIS 2.0, per le applicazioni grafiche 3D, la decodifica MPEG in tempo reale, la compressione dei dati, l'elaborazione dei segnali, l'esecuzione di programmi Java e le connessioni di rete.

Il processore UltraSPARC III, anche se è utilizzato in alcune workstation, è stato inizialmente concepito per essere impiegato nella linea principale di Sun, ovvero i server multiprocessore dotati di una grande memoria condivisa, utilizzati in Internet e nelle reti delle grandi aziende. In particolare gran parte del "collante" necessario per assemblare un multiprocessore è inclusa all'interno del chip UltraSPARC III stesso, rendendo così più facile connettere fra loro un gran numero di processori.

Il primo processore UltraSPARC III, introdotto nel 2000, ha una velocità di 600 MHz, linee di alluminio di 0,18 micron e 29 milioni di transistor. Dato che il volume di vendite di Sun era troppo piccolo per allestire un impianto di produzione allo stato dell'arte si è preferito concentrarsi sulla progettazione del chip e sul software e affidare la fabbricazione del processore ad altri produttori di chip. I chip di UltraSPARC III sono prodotti dalla Texas Instruments. Nel 2000 TI ha migliorato la propria tecnologia e ha cominciato a produrre chip a 900 MHz e 0,15 micron usando collegamenti in rame al posto di quelli in alluminio. Nel 2002 la larghezza della linea è scesa a 0,13 micron e il clock è salito a 1,2 GHz. Questi chip richiedono 50 watt di potenza e presentano quindi gli stessi problemi di dissipazione del calore visti nel caso del Pentium 4.

È difficile paragonare un chip CISC (come il Pentium 4) a un RISC (come l'UltraSPARC III) basandosi solamente sulla velocità di clock. Per esempio, l'UltraSPARC III è in grado di lanciare in continuazione quattro istruzioni per ciclo di clock, fornendo all'incirca la stessa velocità di esecuzione di una CPU che funziona a 4,8 GHz ed emette una sola istruzione per ciclo. L'UltraSPARC ha inoltre sei pipeline interne, che comprendono due pipeline a 14 stadi per le operazioni su interi, due per le operazioni in virgola mobile, una per le operazioni di load/store e una per i salti. Utilizza inoltre un approccio diverso per la cache, impiega bus più ampi e altri fattori ne migliorano ulteriormente le prestazioni. Anche il Pentium 4 ha i propri punti di forza. La questione è che il confronto tra due chip molto diversi basandosi esclusivamente sulle rispettive velocità di clock fornisce scarse informazioni circa le relative prestazioni nell'esecuzione di compiti specifici.

Come si può vedere nella Figura 3.47, l'UltraSPARC III è un Land Grid Array composto da 1368 pin. L'involucro consiste in una matrice quadrata di 37 × 37 = 1369 pin sulla parte inferiore, in cui manca il pin nell'angolo in basso a sinistra, per evitare che possa essere inserito in modo scorretto.

L'UltraSPARC III ha due cache interne principali, di 32 KB per le istruzioni e di 64 per i dati. È presente inoltre una cache di prefetch da 2 KB, e un'ulteriore cache di scrittura da 2 KB è utilizzata per raccogliere le scritture verso la cache di secondo livello, in modo che possano avvenire in raffiche più grandi, migliorando di conseguenza l'uso della banda. L'UltraSPARC, analogamente al Pentium 4, usa una cache di secondo livello esterna al chip, ma questa, diversamente dal Pentium 4, non si trova all'interno dello stesso contenitore del chip. Il controllore della cache e i componenti logici per localizzare i blocchi della cache si trovano sul chip, al contrario della memoria SRAM. In questo modo per la cache di secondo livello i progettisti sono liberi di scegliere uno qualsiasi dei chip di cache disponibili commercialmente.

La scelta di integrare la cache di secondo livello sul Pentium 4 e di tenerla separata sull'UltraSPARC III è dovuta in parte a ragioni tecniche e in parte ai differenti approcci com-

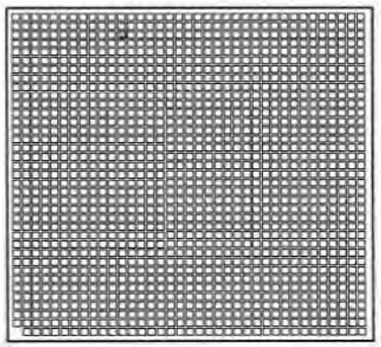


Figura 3.47 Chip della CPU UltraSPARC III.

merciali adottati da Intel e Sun. Dal punto di vista tecnico una cache esterna è più grande e più flessibile (le cache L2 dell'UltraSPARC III variano da 1 a 8 MB, mentre le cache L2 del Pentium 4 hanno una dimensione fissa di 512 KB), ma può risultare più lenta a causa della maggiore distanza dalla CPU. Inoltre è necessario un maggior numero di segnali per indirizzare la cache. In particolare la larghezza della connessione tra l'UltraSPARC III e la sua cache L2 è pari a 256 bit, il che permette di trasferire in un unico ciclo un intero blocco della cache da 32 byte.

Per quanto riguarda gli aspetti commerciali, Intel è un venditore di semiconduttori e ha la possibilità di progettare e produrre i propri chip di cache L2 e di connetterli alla CPU attraverso un'interfaccia proprietaria ad alte prestazioni. Sun, invece, realizza calcolatori, non chip. Sun progetta alcuni dei propri chip (come gli UltraSPARC), ma delega la loro realizzazione ai produttori di semiconduttori. Quando è possibile Sun preferisce utilizzare chip disponibili commercialmente e che sono stati raffinati scrupolosamente, provenendo da mercati competitivi. Dato che numerosi produttori forniscono le SRAM impiegate nelle cache di secondo livello, Sun non ha una particolare necessità di progettarne una propria. Questa scelta implica che la cache di secondo livello è indipendente dal chip della CPU.

Il bus indirizzi dell'UltraSPARC III è largo 43 bit e consente di gestire fino a 8 TB di memoria centrale. Il bus dati è largo 128 bit e permette il trasferimento parallelo, tra la CPU e la memoria, di 16 byte. La velocità del bus è di 150 MHz, corrispondente a una larghezza di banda di 2,4 GB/s, molto più veloce di quella del bus PCI che è pari a 528 MB/s.

Per consentire che più CPU UltraSPARC comunichino con più memorie, Sun ha sviluppato **UPA** (*Ultra Port Architecture*), realizzabile come un bus, un commutatore o come entrambi. Diversi server e workstation utilizzano differenti implementazioni di UPA. Per la CPU, l'implementazione di UPA non è importante, dato che essa deve supportare (e lo fa) solamente l'interfaccia, definita in modo preciso.

Nella Figura 3.48 è rappresentato il nucleo di un sistema UltraSPARC III: il chip della CPU, l'interfaccia UPA e la cache di secondo livello (due SRAM commerciali). Nella figura è compreso anche un chip **UDB II** (*UltraSPARC Data Buffer II*), di cui tra poco descriveremo la funzione. Per richiedere una parola della memoria, La CPU la cerca inizialmente in una delle proprie cache (interne) di primo livello. Se la trova, continua l'esecuzione alla massima velocità, altrimenti prova a cercarla in quella di secondo livello.

Anche se nel corso del Capitolo 4 tratteremo le cache in maggior dettaglio è utile spendere ora alcune parole al riguardo. Tutta la memoria centrale è suddivisa in linee di cache (blocchi) di 64 byte. All'interno della cache di primo livello sono mantenute le 256 linee d'istruzioni e le 256 linee di dati maggiormente utilizzate. Le linee di cache usate frequentemente, ma che non trovano spazio nella cache di primo livello, vengono memorizzate in quella di secondo livello. Questa cache contiene dati e istruzioni, mischiati fra loro senza alcun ordine particolare e memorizzati nel rettangolo etichettato con "Cache di dati di secondo livello". Il sistema deve tener traccia di quali linee si trovano nella cache di secondo livello; questa informazione è mantenuta all'interno della seconda SRAM, etichettata con "Cache dei tag di secondo livello".

Quando si verifica un fallimento (*miss*) nella cache di primo livello la CPU spedisce l'identificatore della linea che sta cercando (tag dell'indirizzo) alla cache di secondo livello. La risposta (tag dei dati) comunica alla CPU se la linea si trova all'interno della cache di

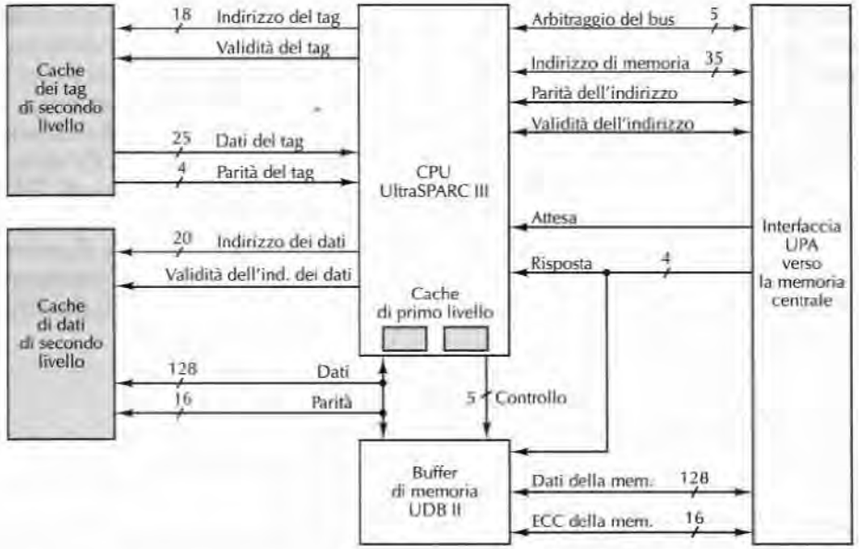


Figura 3.48 Caratteristiche principali del nucleo di un sistema UltraSPARC III.

secondo livello e, in tal caso, in quale stato si trova. Se la linea è memorizzata nella cache la CPU può ottenerla. I dati sono trasferiti a 16 byte per volta e servono dunque quattro cicli per prelevare un'intera linea e inserirla nella cache di primo livello.

Se la linea di cache non si trova neanche nella cache di secondo livello, occorre prelevarla dalla memoria centrale attraverso l'interfaccia UPA. L'UPA di UltraSPARC è implementata come un controllore centralizzato al quale giungono gli indirizzi e i segnali di controllo dalla CPU (o di tutte le CPU se ce n'è più di una). Per accedere alla memoria, la CPU deve inizialmente utilizzare i pin per l'arbitraggio del bus, in modo da acquisire il diritto a proseguire. Una volta che le è stato concesso il permesso, la CPU produce un output sui pin dell'indirizzo di memoria, specifica il tipo di richiesta e asserisce il pin per la validità dell'indirizzo. Questi pin sono bidirezionali dato che in un sistema multiprocessore altre CPU devono poter accedere alle cache remote per far sì che siano tutte consistenti. L'indirizzo e il tipo di ciclo di bus sono spediti sui pin d'indirizzo in due cicli; come abbiamo visto nella Figura 3.31 la riga viene spedita durante il primo ciclo, mentre la colonna durante il secondo.

In attesa dei risultati, la CPU può continuare a svolgere il proprio lavoro. Un fallimento della cache durante il prefetching di un'istruzione non impedisce, per esempio, l'esecuzione di una o più istruzioni già prelevate, ciascuna delle quali potrebbe far riferimento a dati non presenti in alcuna cache. Nello stesso momento potrebbero quindi verificarsi più transazioni verso l'UPA. Questo controllore può gestire due flussi di transazioni indipendenti (in genere letture e scritture), ciascuna con più transazioni in attesa. È compito del controllore centralizzato tener traccia di tutto ciò ed effettuare le richieste alla memoria nell'ordine più efficiente.

I dati possono arrivare alla memoria in gruppi di 8 byte e sono associati a un codice a correzione di errore di 16 bit per garantire una maggiore affidabilità. Una transazione potrebbe richiedere un intero blocco della cache, una *quadword* (8 byte) oppure pochi byte. Tutti i dati provenienti dal sistema di memoria giungono all'UDB, che li memorizza temporaneamente. Questo buffer disaccoppia ulteriormente la CPU dal sistema di memoria cosicché entrambi possano funzionare in modo asincrono. Se per esempio la CPU deve scrivere una parola o una linea di cache in memoria, invece di attendere l'accesso all'UPA può scrivere immediatamente i dati nell'UDB e lasciare che sia quest'ultimo a gestirne successivamente il prelievo dalla memoria. L'UDB genera inoltre codici a correzione di errore e li controlla. Bisogna precisare che, come nel caso del Pentium 4, anche la descrizione di UltraSPARC III è stata fortemente semplificata; ciononostante si è cercato di descrivere la funzione principale delle operazioni.

3.5.3 Chip 8051

Sia il Pentium 4 sia l'UltraSPARC III sono esempi di CPU ad alte prestazioni progettate per costruire PC e server estremamente veloci. Quando si pensa ai calcolatori si ha in mente questo tipo di sistemi. Esiste tuttavia un altro, intero, mondo di calcolatori che è effettivamente ancora più vasto: i sistemi integrati. In questo paragrafo daremo un rapido sguardo a questo mondo.

Forse è un po' esagerato affermare che ogni dispositivo elettrico dal costo superiore ai 100 euro contiene al suo interno un calcolatore. Tuttavia al giorno d'oggi televisori, telefoni cellulari, agende elettroniche, forni a microonde, videocamere digitali, videoregistratori, stampanti laser, allarmi antifurto, protesi acustiche, giochi elettronici e tanti altri dispositivi sono controllati mediante calcolatori. Di solito si tende a rendere più economici i calcolatori inseriti all'interno di questi oggetti piuttosto che dotarli di elevate prestazioni; ciò porta a compromessi diversi rispetto a quelli che si compiono per le CPU ad alte prestazioni studiate finora.

Come già detto nel Capitolo 1, l'8051 è probabilmente il microcontrollore più diffuso tra quelli oggi in uso; il motivo è da ricercare principalmente nel suo basso costo. Come vedremo tra poco è oltretutto un chip particolarmente semplice che rende il collegamento agevole e poco costoso. Passiamo allora a esaminare l'8051, la cui disposizione fisica dei contatti è mostrata nella Figura 3.49.

L'usuale contenitore dell'8051 ha 40 pin (anche se sono possibili delle varianti per usi specifici). Vi sono 16 linee d'indirizzo e quindi è possibile indirizzare fino a 64 KB di memoria. Il bus dei dati è largo 8 bit e quindi i trasferimenti dati tra la CPU e la memoria avvengono un byte alla volta (contro gli 8 byte alla volta del Pentium 4 e i 16 byte alla volta di UltraSPARC III). Ci sono varie linee di controllo, descritte nel seguito, ma la differenza più rilevante rispetto al Pentium 4 e all'UltraSPARC, che sono pure CPU, è la presenza di 32 linee di I/O, organizzate in quattro gruppi di 8 bit ciascuno. A ognuna di queste linee è possibile collegare un pulsante, un interruttore, un LED oppure altri dispositivi che possano fornire un input o un output all'8051. In una radiosveglia ciascuno dei pulsanti e degli interruttori potrebbe per esempio essere collegato a una differente linea di I/O, e altre linee di I/O potrebbero controllare lo schermo. In questo modo la maggior parte delle funzioni, se non tutte, potrebbe essere controllata via software eliminando il costo di altre componenti logiche.

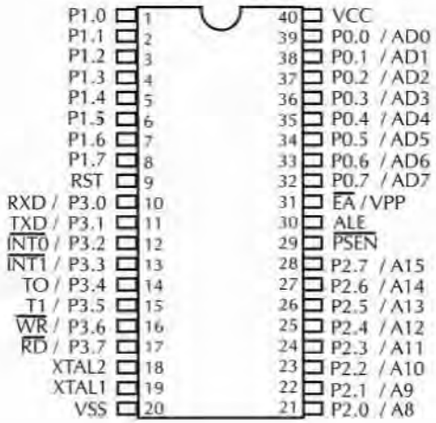


Figura 3.49 Disposizione fisica dei contatti dell'8051.

La Figura 3.50 mostra la disposizione logica dei contatti dell'8051. Questo controllore ha 4 KB di ROM interna (8 KB nell'8052), e se tale quantità è insufficiente è possibile collegare al bus dell'8051 fino a 64 KB di memoria esterna. I primi sette segnali sul lato sinistro della Figura 3.50 sono utilizzati per interfacciarsi a eventuali memorie esterne. Il primo segnale, A, contiene 16 linee d'indirizzo per indirizzare il byte della memoria esterna che si intende leggere o scrivere. Le otto linee D sono invece utilizzate per il trasporto dei dati. Le 8 linee d'indirizzo meno significative sono multiplexate negli stessi pin delle linee dei dati per

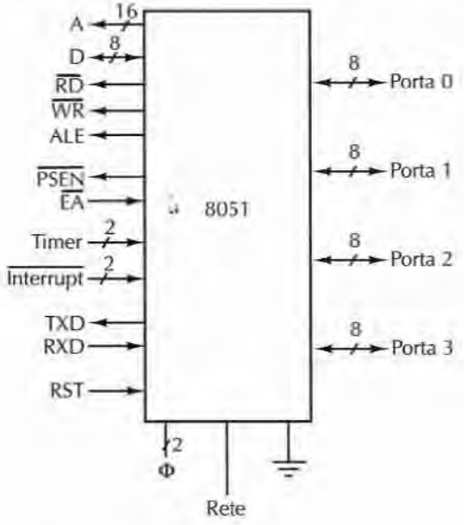


Figura 3.50 Disposizione logica dei contatti dell'8051.

ridurre il numero totale di pin. Durante una transazione del bus nel primo ciclo questi pin mandano in output l'indirizzo, mentre nei successivi trasportano i dati.

Quando si utilizza una memoria esterna, l'8051 deve indicare se sta leggendo o scrivendo la memoria asserendo rispettivamente $\overline{\text{RD}}$ oppure $\overline{\text{WR}}$. Il segnale ALE (*Address Latch Enable*) è utilizzato in presenza di una memoria esterna. La CPU asserisce questo segnale per indicare che l'indirizzo è valido. Le memorie esterne lo utilizzano generalmente per stabilizzare le linee degi indirizzi, affinché possano essere rilasciate poco dopo in modo che i pin siano riutilizzabili per i dati.

Anche $\overline{\text{PSEN}}$ e $\overline{\text{EA\#}}$ riguardano la memoria esterna. Il segnale $\overline{\text{PSEN}}$ (*Program Store ENable*) viene asserito per indicare che l'8051 vuole leggere dalla memoria contenente il programma. In genere dovrebbe essere collegato al segnale OE della memoria, com'è illustrato nella Figura 3.29.

Il segnale $\overline{\text{EA\#}}$ (*External Access*) è generalmente collegato al valore alto o basso, in modo che abbia sempre lo stesso valore. Se è collegato al valore alto la memoria interna di 4 KB (8 KB nell'8052) è utilizzata per gli indirizzi interni e la memoria esterna per gli indirizzi oltre i 4 KB (8 KB nell'8052). Se invece è collegato al valore basso la memoria esterna è utilizzata per tutti gli indirizzi e la memoria sul chip viene in realtà scavalcata. Sull'8031 e sull'8032 $\overline{\text{EA\#}}$ deve essere collegato al valore basso, dato che non c'è memoria sul chip.

Le due linee dei timer permettono di usare timer esterni come input della CPU, mentre le due linee di interrupt consentono a due diversi dispositivi esterni di interrompere la CPU. Le linee TXD e RXD sono presenti per consentire I/O seriale verso un terminale o un modem. Infine la linea RST consente all'utente o a componenti hardware esterni di resettare l'8051; questa linea è in genere asserita quando qualcosa va storto e il sistema deve essere reinizializzato.

L'8051 è simile alla maggior parte delle CPU a 8 bit tranne per la presenza delle linee per l'I/O seriale. Ciò che distingue l'8051 è la presenza di 32 linee di I/O, organizzate come quattro porte e mostrate sul lato destro della Figura 3.50. Ciascuna di queste linee è bidirezionale e può essere letta e scritta dal programma. Questa è la principale forma d'interazione tra l'8051 e il mondo esterno e che lo rende allo stesso tempo così valido: un unico chip è tutto ciò che serve per la CPU, la memoria e l'I/O.

3.6 Esempi di bus

I bus sono il collante che tiene insieme i sistemi di calcolatori. In questo paragrafo daremo uno sguardo dettagliato ad alcuni modelli molto diffusi: il bus ISA, il bus PCI e l'Universal Serial Bus. Il bus ISA è una leggera espansione del bus del PC IBM originario. Fino a pochi anni fa era ancora presente sui PC basati su Intel per ragioni di retrocompatibilità. In seguito Intel e Microsoft hanno deciso di eliminarlo. In ogni caso da alcuni anni i PC hanno anche un secondo e più veloce bus: il PCI. Questo bus è più largo del bus ISA e funziona con un clock più elevato. È quindi in grado di trasportare una quantità maggiore di dati al secondo rispetto al bus ISA. Per la maggior parte dei PC attuali è un vero e proprio cavallo da tiro, anche se si comincia già a intravedere il suo successore. L'Universal Serial Bus è un bus per periferiche a bassa velocità, come mouse e tastiere, che sta guadagnando sempre più popolarità; una seconda versione del bus USB funziona a velocità decisamente più elevate. Nei paragrafi successivi analizzeremo questi bus per scoprire come funzionano.

3.6.1 Bus ISA

Il bus del PC IBM è stato lo standard de facto dei sistemi basati su 8088; infatti quasi tutti i venditori di cloni di PC lo copiarono per permettere di utilizzare anche con i loro sistemi le numerose schede di I/O prodotte da terze parti. Questo bus aveva 62 linee di segnali, tra cui 20 per gli indirizzi della memoria, 8 per i dati e una per asserire rispettivamente lettura e scrittura in memoria e lettura e scrittura sulle periferiche. Erano presenti inoltre segnali per richiedere e concedere gli interrupt e per utilizzare il DMA. In sostanza si trattava di un bus molto semplice.

Fisicamente ISA era collegato alla scheda madre del PC con circa mezza dozzina di connettori distanziati 2 cm l'uno dall'altro, nei quali potevano essere inserite delle schede; ciascuna di loro aveva un'aletta che si adattava perfettamente al connettore. L'aletta presentava su ciascun lato 32 strisce placcate oro che permettevano il contatto elettrico con il connettore.

Quando IBM introdusse il PC/AT basato sul processore 80286, si trovò tra le mani un problema. Se avesse cominciato da zero un nuovo progetto per un bus a 16 bit molti potenziali clienti avrebbero esitato ad acquistarlo, poiché nessuna delle numerose schede aggiuntive per PC avrebbe potuto funzionare con la nuova macchina. D'altra parte, se fosse rimasta legata al bus del PC, alle sue 20 linee d'indirizzi e alle sue 8 linee di dati, non avrebbe potuto trarre vantaggio dalle nuove capacità dell'80286, come quella d'indirizzare 16 MB di memoria e trasferire parole di 16 bit.

La soluzione scelta fu quella di estendere il bus del PC. Le schede aggiuntive per PC avevano un connettore laterale con 62 contatti, ma questo connettore non si estendeva lungo tutta la larghezza della scheda. La soluzione adottata dal PC/AT fu quella di aggiungere sulla parte inferiore della scheda un secondo connettore laterale vicino a quello principale e di progettare l'AT in modo da funzionare con entrambi i tipi di schede. L'idea è illustrata a grandi linee nella Figura 3.51.

Il secondo connettore sul bus del PC/AT contiene 36 linee. Fra queste, 31 servono a fornire ulteriori linee d'indirizzo, di dati, di interrupt e canali DMA, oltre alla tensione e alla terra. Le restanti servono invece a gestire le differenze tra i trasferimenti a 8 bit e quelli a 16 bit.

Quando IBM decise di fare della serie PS/2 il successore del PC e del PC/AT, stabilì che era arrivato il momento di ricominciare. Parte di questa decisione fu probabilmente tecnica (il bus del PC era ormai diventato realmente obsoleto), ma senza dubbio fu anche motivata dal desiderio di ostacolare i produttori di cloni che avevano conquistato una fetta del mercato eccessivamente ampia. Le macchine PS/2 di fascia media e alta furono quindi dotate di un bus completamente nuovo, chiamato Microchannel, protetto da una barriera di brevetti sostenuta da una schiera di avvocati.

Il resto del mercato dei personal computer adottò un proprio standard, il bus **ISA** (*Industry Standard Architecture*), che in pratica era il bus del PC/AT funzionante a 8,33 MHz. Il vantaggio principale di questo approccio fu di mantenere la compatibilità tra le macchine e le schede esistenti. Inoltre era basato su un bus che IBM aveva liberamente dato in licenza a diverse società per assicurarsi che molti produttori creassero schede per il PC originario. La situazione si ribaltò contro IBM e finì per allontanarla dal mercato dei PC. Fino a pochi anni fa la maggior parte dei PC basati su Intel aveva ancora questo bus, seppur accompagnato da uno o più bus di tipo diverso.

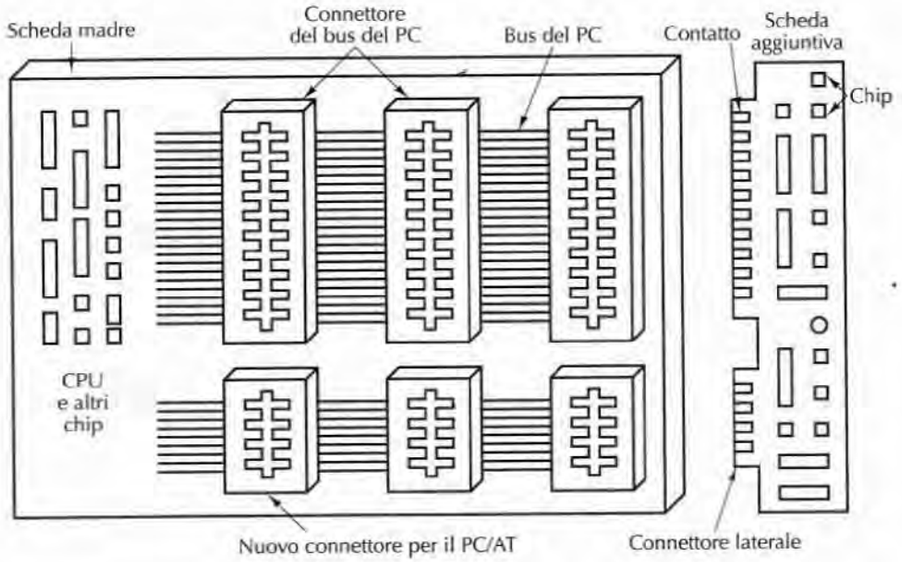


Figura 3.51 I due componenti del bus del PC/AT: una parte uguale a quella del PC originario e una parte nuova.

Successivamente il bus ISA fu esteso a 32 bit con l'aggiunta di alcune nuove funzionalità (per esempio quelle per i multiprocessori) e prese il nome di **EISA** (*Extended ISA*).

3.6.2 Bus PCI

Sul PC IBM originario la maggior parte delle applicazioni era di tipo testuale. Gradualmente, con l'introduzione di Windows, presero piede le interfacce grafiche per l'utente. Nessuna di queste applicazioni richiedeva al bus ISA un eccessivo sforzo. Tuttavia la situazione cambiò radicalmente nel momento in cui molte applicazioni, in particolare i giochi multimediali, cominciarono a utilizzare il calcolatore per visualizzare immagini a pieno schermo e in movimento.

Facciamo un semplice calcolo. Consideriamo uno schermo con la risoluzione di 1024 × 768 utilizzato per immagini in movimento e in *true color* (3 byte/pixel): ogni schermata contiene 2,25 MB di dati. Per ottenere un movimento senza scatti sono necessarie almeno 30 schermate al secondo per un tasso di trasferimenti dati complessivo di 67,5 MB/s. La situazione in realtà è ancora peggiore, dato che per visualizzare un video da un hard disk, da un CD-ROM o da un DVD i dati devono passare dal lettore del disco alla memoria lungo il bus. Successivamente, per essere visualizzati, i dati devono viaggiare nuovamente sul bus per raggiungere l'adattatore grafico. È necessaria quindi una larghezza di banda di 135 MB/s solo per il video, senza considerare la larghezza di banda per la CPU e altri dispositivi.

Il bus ISA aveva una larghezza di banda massima di 16,7 MB/s, determinata da una velocità di 8,33 MHz e dalla capacità di trasferire 2 byte per ciclo; il bus EISA poteva invece spostare 4 byte per ciclo, raggiungendo così 33,3 MB/s. Nessuno di questi due bus si avvicinava alla larghezza di banda necessaria per visualizzare video a schermo intero.

Nel 1990 Intel colse questa necessità e progettò un nuovo bus con una larghezza di banda molto più ampia rispetto allo stesso bus EISA e lo chiamò **PCI** (*Peripheral Component Interconnect*). Per incoraggiarne l'adozione, Intel brevettò il bus PCI e poi rese di pubblico dominio i brevetti, in modo che qualunque azienda potesse costruire delle periferiche per il bus senza pagare alcun diritto. Intel fondò inoltre un consorzio, chiamato *PCI Special Interest Group* per la gestione del futuro del bus. La conseguenza di queste azioni fu l'incremento della sua diffusione. A partire dal Pentium, praticamente ogni calcolatore basato su Intel, così come molti altri computer, aveva un bus PCI. Anche Sun ha una versione di UltraSPARC, l'UltraSPARC IIIi, che utilizza il bus PCI. Shanley e Anderson (1999) e Solari e Willse (2004) trattano il bus PCI molto approfonditamente.

Il bus PCI originario trasferiva 32 bit per ciclo e funzionava a 33 MHz (tempo di ciclo di 30 ns) per una larghezza di banda totale di 133 MB/s. Nel 1993 fu introdotto il PCI 2.0 e nel 1995 il PCI 2.1. La versione PCI 2.2 ha invece delle funzionalità specifiche per i calcolatori portatili (la maggior parte delle quali serve a risparmiare energia). Il bus PCI può funzionare fino a 66 MHz e può gestire trasferimenti di 64 bit, per una larghezza di banda totale di 528 MB/s; con questa banda è possibile (assumendo che il disco e il resto del sistema siano pronti per il compito) visualizzare in modo fluido i video a tutto schermo.

Anche se 528 MB/s suonano come una velocità apprezzabile, rimanevano tuttavia due problemi. Primo, la larghezza di banda non era sufficiente per essere usata come bus di memoria. Secondo, non era compatibile con tutte le vecchie schede ISA ancora in circolazione. La decisione presa da Intel fu di progettare calcolatori con tre o più bus. Com'è possibile vedere nella Figura 3.52 la CPU può comunicare con la memoria centrale attraverso uno speciale bus di memoria; inoltre è presente un bus ISA collegato al bus PCI. Questa soluzione soddisfaceva tutti i requisiti e per questo motivo nel corso degli anni '90 fu adottata in modo molto diffuso.

In questa architettura le due componenti chiave sono i due chip bridge prodotti da Intel (che aveva quindi un grande interesse nell'intero progetto). Il bridge PCI connette la CPU, la memoria e il bus PCI, mentre il bridge ISA connette il bus PCI al bus ISA e supporta inoltre uno o due dischi IDE. Quasi tutti i sistemi Pentium 4 hanno uno o più slot PCI liberi per aggiungere nuove periferiche ad alta velocità, e uno o più slot ISA per periferiche a bassa velocità.

Il vantaggio principale dell'architettura mostrata nella Figura 3.51 è che la CPU, usando un bus di memoria proprietario, ha una larghezza di banda estremamente alta verso la memoria. Il bus PCI offre invece un'ampia larghezza di banda per periferiche veloci come i dischi SCSI, gli adattatori grafici, ecc.; inoltre, grazie al bus ISA, è ancora possibile utilizzare le vecchie schede. Nella figura è presente un altro riquadro, definito USB, che si riferisce all'Universal Serial Bus e sarà trattato nel capitolo successivo.

Sarebbe stato più pratico avere un solo tipo di schede PCI. Sfortunatamente non è così, dato che vi sono varie possibilità per tensione, ampiezza e temporizzazione.

Il bus PCI supporta due diverse tensioni, dato che i calcolatori più vecchi funzionavano spesso a 5 volt, mentre con quelli più recenti si tende a usare 3,3 volt. I connettori sono gli stessi, tranne per due frammenti di plastica che impediscono all'utente di inserire

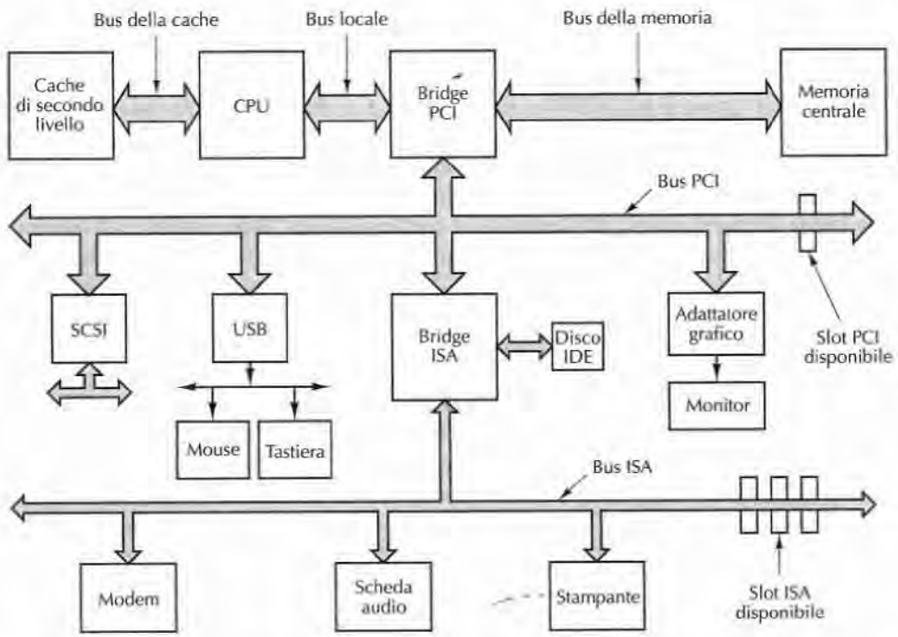


Figura 3.52 Architettura di uno dei primi Pentium. I bus più spessi hanno una maggiore larghezza di banda rispetto a quelli più sottili (la figura non è in scala).

una scheda a 5 volt in un bus PCI a 3,3 volt e viceversa. Fortunatamente esistono schede universali che supportano entrambe le tensioni e che possono essere inserite nei due tipi di slot. Le schede esistono anche in versione 32 bit oppure 64 bit. Quelle a 32 bit hanno 120 pin, e le altre hanno ulteriori 64 pin; l'aggiunta di pin è analoga al modo in cui fu esteso il bus del PC IBM per portarlo a 16 bit (vedi Figura 3.51). Un sistema con bus PCI che supporta schede a 64 bit può accettare anche schede a 32 bit, mentre non è vero il contrario. Infine i bus PCI e le loro schede possono funzionare a 33 oppure a 66 MHz e la scelta viene effettuata collegando un pin all'alimentazione oppure collegandolo alla terra. I connettori sono dello stesso tipo per entrambe le velocità.

Alla fine degli anni '90 quasi tutti furono d'accordo nel considerare il bus ISA ormai superato e decisero di escluderlo dalle nuove architetture. In quel periodo però le risoluzioni dei monitor aumentarono, raggiungendo in alcuni casi 1600 × 1200; inoltre crebbe la richiesta di video fluido e a tutto schermo, in particolar modo nell'area dei giochi interattivi. Intel decise quindi di aggiungere un altro bus per pilotare esclusivamente le schede grafiche. Questo bus prese il nome di **AGP** (*Accelerated Graphics Port*). La versione iniziale, AGP 1.0, funzionava a 264 MB/s. Questo bus, al quale ci si riferisce con 1x, era più lento del PCI, ma in compenso era dedicato solamente alla scheda grafica. Nel corso degli anni apparvero nuove versioni, come l'AGP 3.0 che raggiunse una larghezza di banda di 2,1 GB/s (8x). La Figura 3.53 mostra un moderno sistema basato su Pentium 4.

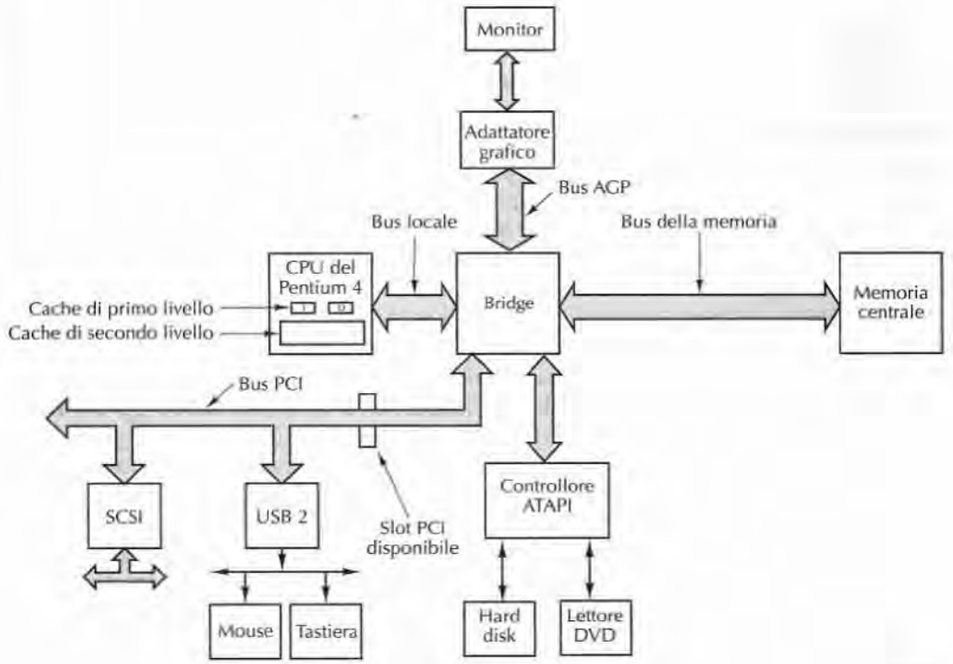


Figura 3.53 Struttura del bus di un moderno Pentium 4.

In questa architettura diventa centrale il chip bridge, che collega le cinque principali parti del sistema: CPU, memoria, scheda grafica, controllore ATAPI e bus PCI. In alcune versioni supporta anche Ethernet e altri dispositivi ad alta velocità. I dispositivi a bassa velocità sono invece collegati al bus PCI.

Al suo interno il chip bridge è diviso in due parti: quella verso la memoria e quella verso l'I/O. Il primo connette la CPU alla memoria e all'adattatore grafico e il secondo connette, mediante una connessione diretta verso ognuno di loro, il controllore ATAPI, il bus PCI ed eventualmente altri dispositivi veloci di I/O.

Anche il bus PCI, come tutti quelli successivi al bus del PC IBM originario, è sincrono. Tutte le transazioni sul bus PCI avvengono tra un master, detto convenzionalmente **iniziatore**, e uno slave, chiamato convenzionalmente **destinatario**. Inoltre, per ridurre il numero totale di pin, le linee degli indirizzi e dei dati sono multiplexate. In questo modo sulle schede PCI sono necessari soltanto 64 pin per i segnali d'indirizzo e di dati, anche se PCI supporta sia indirizzi sia dati a 64 bit.

I pin multiplexati per gli indirizzi e per i dati funzionano nel modo seguente. Nel ciclo 1 di un'operazione di lettura il master immette l'indirizzo sul bus. Nel ciclo 2 il master rimuove l'indirizzo e il bus è invertito in modo che lo slave lo possa utilizzare; infine, nel ciclo 3, lo slave genera in output i dati richiesti. Nelle operazioni di scrittura il bus non deve essere invertito, dato che è il master a fornire sia l'indirizzo sia i dati. Ciononostante la tran-

sazione minima rimane tuttavia della durata di 3 cicli. Se non è in grado di rispondere nei tre cicli, lo slave può inserire degli stati di attesa. Sono inoltre consentiti trasferimenti di blocchi di dimensione illimitata, e diversi tipi di cicli di bus.

Arbitraggio del bus PCI

Prima di poter utilizzare il bus PCI, i dispositivi devono acquisirlo. Come mostrato nella Figura 3.54 si utilizza un arbitro centralizzato. Nella maggior parte delle architetture l'arbitro del bus è integrato in uno dei chip bridge. Ogni dispositivo PCI ha due linee dedicate che lo collegano all'arbitro: una, REQ#, per richiedere il bus, e l'altra, GNT#, per riceverne la concessione.

Per richiedere il bus un dispositivo PCI (compresa la CPU) deve asserire REQ# e attendere finché non veda che la linea GNT# è stata asserita dall'arbitro. Quando ciò si verifica il dispositivo può utilizzare il bus durante il ciclo successivo. Le specifiche PCI non definiscono l'algoritmo che l'arbitro deve utilizzare; sono consentiti arbitraggi round-robin, arbitraggi basati su priorità e altre politiche. Ovviamente un arbitro dovrebbe essere imparziale in modo che nessun dispositivo sia obbligato ad attendere all'infinito la concessione del bus.

Il bus viene concesso per una transazione alla volta, anche se può avere lunghezza teoricamente illimitata. Se un dispositivo vuole effettuare una seconda transazione e nessun altro dispositivo sta richiedendo il bus, allora può continuare a utilizzarlo; tuttavia tra la prima e la seconda transazione deve spesso essere inserito un ciclo di inattività. In particolari circostante e in assenza di contesa per l'utilizzo del bus, un dispositivo può tuttavia effettuare transazioni in sequenza senza dover inserire un ciclo di inattività. L'arbitro può negare la linea GNT# nel caso in cui un master stia effettuando un trasferimento molto lungo e contemporaneamente qualche altro dispositivo richieda il bus. Dato che il master del bus deve monitorare la linea GNT#, esso deve rilasciare il bus al ciclo successivo non appena vede che la linea è stata negata. Questo schema permette trasferimenti molto lunghi (che sono efficienti) quando c'è un solo candidato master, ma allo stesso tempo continua a garantire una risposta veloce quando più dispositivi competono per l'utilizzo del bus.

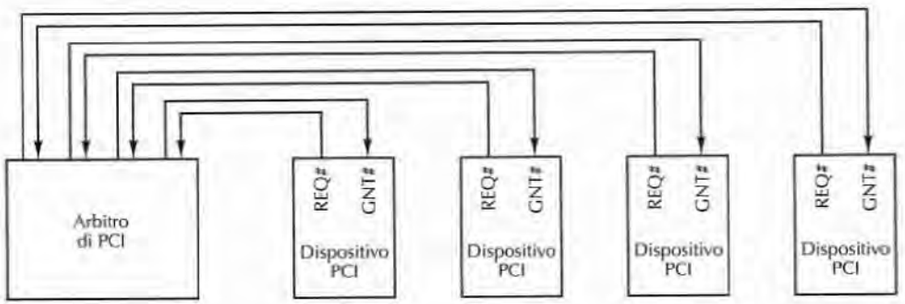


Figura 3.54 Il bus PCI utilizza un arbitro del bus centralizzato.

Segnali del bus PCI

Nel bus PCI ci sono dei segnali obbligatori, mostrati nella Figura 3.55(a), e dei segnali opzionali, mostrati nella Figura 3.55(b). I pin rimanenti dei 120 (o 184) totali sono utilizzati per la tensione, la terra e altre funzioni che non elencheremo. Le colonne *Master* (iniziatore)

Segnale	Linee	Master	Slave	Descrizione
CLK	1			Clock (33 MHz oppure 66 MHz)
AD	32	×	×	Linee d'indirizzo e dei dati multiplexate
PAR	1	×		Bit di parità dell'indirizzo e dei dati
C/BE	4	×		Comando del bus/bit map per i byte abilitati
FRAME#	1	×		Indica che AD e C/BE sono asserite
IRDY#	1	×		Lettura: il master accetterà i dati; scrittura: i dati sono presenti
IDSEL	1	×		Seleziona lo spazio di configurazione invece che quello di memoria
DEVSEL#	1		×	Lo slave ha decodificato il proprio indirizzo ed è in ascolto
TRDY#	1		×	Lettura: i dati sono presenti; scrittura: lo slave accetterà i dati
STOP#	1		×	Lo slave vuole interrompere immediatamente la transazione
PERR#	1			Il destinatario ha rilevato un errore sulla parità dei dati
SERR#	1			Errore sulla parità dell'indirizzo ed errore di sistema
REQ#	1			Arbitraggio del bus: richiesta di appropriazione del bus
GNT#	1			Arbitraggio del bus: concessione del bus
RST#	1			Reinizializza il sistema e tutti i dispositivi

(a)

Segnale	Linee	Master	Slave	Descrizione
REQ64#	1	×		Richiesta di eseguire una transazione a 64 bit
ACK64#	1		×	Concessione per eseguire una transazione a 64 bit
AD	32	×		32 bit aggiuntivi per l'indirizzo o per i dati
PAR64	1	×		Parità per i 32 bit aggiuntivi (indirizzo/dati)
C/BE#	4	×		4 bit aggiuntivi per l'abilitazione dei byte
LOCK	1	×		Blocco del bus per permettere transazioni multiple
SBO#	1			Successo di una cache remota (per un multiprocessore)
SDONE	1			È stato effettuato lo snooping (per un multiprocessore)
INTx	4			Richiesta di un interrupt
JTAG	5			Segnali per i test JTAG IEEE 1149.1
M66EN	1			Collegato all'alimentazione o terra (66 o 33 MHz)

(b)

Figura 3.55 Segnali (a) obbligatori e (b) opzionali del bus PCI.

e *Slave* (destinatario) indicano chi asserisce il segnale su una normale transazione; nel caso in cui il segnale sia asserito da un altro dispositivo (per esempio, CLK) entrambe le colonne sono lasciate vuote.

Analizziamo ora, seppur brevemente, i segnali del bus PCI, iniziando con quelli obbligatori. Il segnale CLK pilota il bus ed è in sincronia con la maggior parte degli altri segnali. Diversamente dal bus ISA, le transazioni del bus PCI iniziano in corrispondenza del fronte di discesa di CLK, che si trova nel mezzo del ciclo invece che all'inizio.

I 32 segnali AD sono destinati agli indirizzi e ai dati (per le transazioni a 32 bit). In genere l'indirizzo viene asserito durante il ciclo 1, mentre i dati durante il ciclo 3. Il segnale PAR è un segnale di parità per AD. Il segnale C/BE# è utilizzato per due scopi distinti; nel ciclo 1 contiene il comando del bus (lettura di una parola, blocco, ecc.), mentre nel ciclo 2 contiene una stringa di 4 bit che indica quali byte di una parola a 32 bit sono validi. Usando C/BE# è possibile leggere oppure scrivere arbitrariamente 1, 2, 3 byte oppure una parola intera.

Il segnale FRAME# viene asserito dal master per iniziare una transazione sul bus comunicando allo slave che l'indirizzo e i comandi del bus sono validi. Di solito, nelle letture, IRDY# viene asserito nello stesso momento di FRAME#; esso comunica che il master è pronto ad accettare dati in ingresso. Nel caso di una scrittura, IRDY# viene invece asserito in un secondo momento, quando i dati si trovano sul bus.

Il segnale IDSEL è legato al fatto che ogni dispositivo PCI deve avere uno spazio di configurazione di 256 byte per le informazioni sulle sue proprietà e che ogni altro dispositivo può leggere (asserendo IDSEL). La proprietà Plug-and-Play di alcuni sistemi operativi utilizza questo spazio di configurazione per determinare quali dispositivi sono presenti sul bus.

Passiamo ora ai segnali asseriti dallo slave. Il primo di questi, DEVSEL#, annuncia che lo slave ha rilevato il proprio indirizzo sulle linee AD ed è pronto a instaurare una transazione. Se DEVSEL# non viene asserito entro un certo intervallo di tempo il master considera scaduto il tempo e assume che il dispositivo indirizzato è assente oppure è guasto.

Il secondo segnale per lo slave, TRDY#, viene asserito nelle letture per annunciare che i dati sono disponibili sulle linee AD, mentre nelle scritture per comunicare che è pronto ad accettare i dati.

I tre segnali successivi servono a notificare un errore. Il primo è STOP#: lo slave lo asserisce se si verifica qualcosa di disastroso e se desidera cancellare la transazione corrente. Il successivo, PERR#, è utilizzato per notificare che nel ciclo precedente si è verificato un errore sulla parità dei dati. In lettura viene asserito dal master, e in scrittura dallo slave. Infine, SERR# serve a notificare errori di sistema o riguardanti gli indirizzi.

I segnali REQ# e GNT# sono utilizzati per l'arbitraggio del bus e non sono asseriti dal master corrente, ma dal dispositivo che intende diventare master del bus. L'ultimo segnale obbligatorio è RST#, utilizzato per resettare il sistema quando l'utente preme il pulsante RESET oppure quando un dispositivo notifica un errore irrimediabile. Quando questo segnale viene asserito tutti i dispositivi vengono resettati e il calcolatore viene riavviato.

Passiamo ora ai segnali opzionali, la maggior parte dei quali riguarda l'espansione da 32 a 64 bit. I segnali REQ64# e ACK64# consentono al master di richiedere l'autorizzazione a effettuare una transazione a 64 bit, e allo slave di accettarla. I segnali AD, PAR64 e C/BE# sono semplicemente un'estensione dei corrispondenti segnali a 32 bit.

I tre segnali successivi non riguardano il passaggio da 32 a 64 bit, ma i sistemi multiprocessore, una funzionalità che le schede PCI non sono obbligate a dover supportare. Il segnale LOCK permette di bloccare il bus per transazioni multiple, mentre i due successivi riguardano lo snooping necessario per mantenere la coerenza della cache.

I segnali INTx sono utilizzati per richiedere interrupt. Sulle schede PCI ci possono essere fino a quattro dispositivi logici distinti e ciascuno di loro può avere la propria linea per la richiesta di interrupt. Il segnale JTAG serve per la procedura di test JTAG IEEE 1149.1. Infine il segnale M66EN è collegato a un valore alto o basso per impostare la velocità di clock e non deve essere modificato mentre il sistema è in funzione.

Transazioni del bus PCI

Il bus PCI (come in genere tutti i bus) è in realtà molto semplice. Per entrare ancora più in sintonia con il suo funzionamento consideriamo il diagramma di temporizzazione della Figura 3.56 che mostra una transazione di lettura, seguita da un ciclo di inattività, seguito a sua volta da una transazione di scrittura, effettuate tutte dallo stesso master del bus.

Quando durante T_1 si verifica il fronte di discesa del clock, il master inserisce l'indirizzo di memoria su AD e il comando del bus su C/BE#; asserisce inoltre FRAME# per iniziare la transazione sul bus.

Durante T_2 il master rilascia l'indirizzo del bus per permettere che venga invertito in modo che lo slave possa pilotarlo durante T_3. Il master modifica inoltre C/BE# per indicare quali byte della parola indirizzata vuole abilitare (cioè leggere).

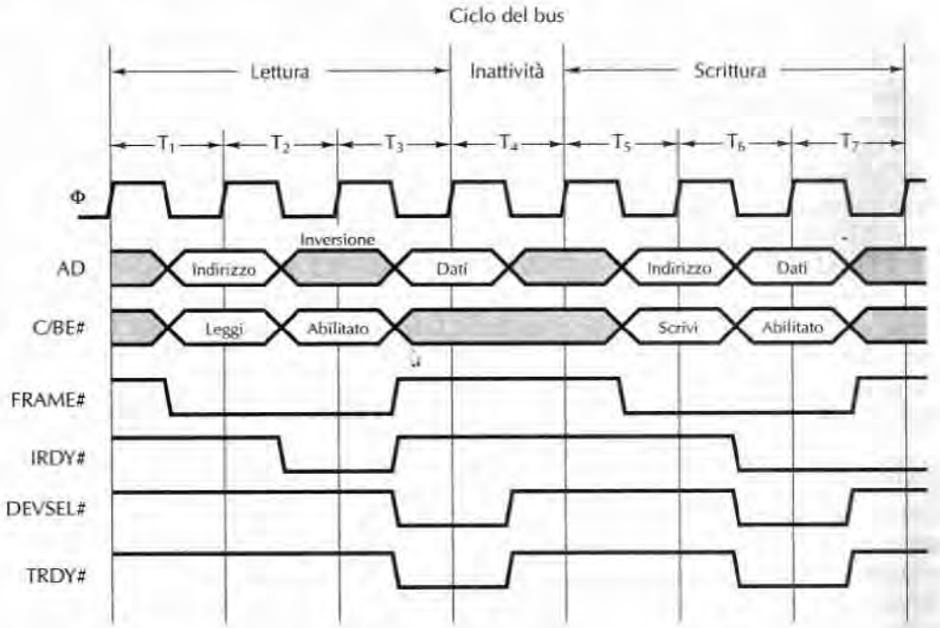


Figura 3.56 Esempi di transazioni di un bus PCI a 32 bit. I primi tre cicli sono utilizzati per un'operazione di lettura. Segue un ciclo d'inattività e infine tre cicli per un'operazione di scrittura.

Durante T_3 lo slave asserisce DEVSEL# in modo che il master sappia che ha rilevato il proprio indirizzo e che si sta adoperando per rispondere. Inoltre scrive i dati sulle linee AD e asserisce TRDY# per notificare il fatto al master. Se lo slave non è in grado di rispondere velocemente deve tuttavia asserire DEVSEL# per annunciare la propria presenza, ma deve lasciare TRDY# negato finché non possa fornire i dati richiesti. Questa procedura potrebbe inserire uno più stati di attesa. In questo esempio (e spesso nelle situazioni reali) il ciclo successivo è inattivo. All'inizio di T_5 vediamo lo stesso master iniziare una scrittura. Come al solito inizia scrivendo sul bus l'indirizzo e il comando del bus. In questo caso però asserisce i dati già nel secondo ciclo; infatti, dato che è lo stesso dispositivo a pilotare le linee AD, non c'è bisogno di un ciclo per l'inversione del bus. In T_7 la memoria accetta i dati.

3.6.3 PCI Express

Anche se il bus PCI funziona in modo adeguato per la maggior parte delle applicazioni attuali, la richiesta di una maggior larghezza di banda per l'I/O sta creando confusione nell'architettura dei PC, un tempo più ordinata. Dalla Figura 3.53 risulta chiaro che il bus PCI non è più l'elemento centrale che tiene unite le varie parti del PC. Parte di quel ruolo è stato preso dal chip bridge.

Il cuore del problema è che ci sono sempre più dispositivi di I/O la cui velocità è eccessiva per il bus PCI. Spingere ulteriormente la frequenza di clock non è una buona idea, perché i problemi di disallineamento del bus, interferenze tra i cavi ed effetti di capacitanza renderebbero la situazione ancora peggiore. Ogni volta che un dispositivo di I/O è diventato troppo veloce per il bus PCI (come le schede grafiche, gli hard disk, le reti, ecc.), Intel ha aggiunto al chip bridge una nuova e speciale porta per permettere a quel dispositivo di scavalcare il bus PCI. Ovviamente questa non può essere una soluzione a lungo termine.

Un ulteriore problema del bus PCI è che le schede sono piuttosto grandi e non possono entrare in un calcolatore portatile o in un palmare, mentre i produttori vorrebbero produrre dispositivi ancora più piccoli. Alcuni produttori vorrebbero inoltre separare i componenti del PC, inserendo per esempio la CPU e la memoria in un piccolo contenitore sigillato e l'hard disk all'interno del monitor. Con le schede PCI questo non è possibile.

Sono state proposte varie soluzioni, ma quella che sembra avere più possibilità di successo (possibilità rilevanti, dato che ci sta dietro Intel) è chiamata **PCI Express**. Questo prodotto ha poco a che fare con il bus PCI, e tra l'altro non è nemmeno un vero e proprio bus, ma gli addetti al marketing non volevano lasciarsi sfuggire un nome così conosciuto come PCI. Scopriamo ora come funziona.

Architettura di PCI Express

Il cuore della soluzione PCI Express è la rinuncia al bus parallelo con i suoi numerosi master e slave e la scelta di un'architettura basata su connessioni seriali punto-a-punto ad alta velocità. Questa soluzione rappresenta un radicale elemento di rottura con la tradizione ISA/EISA/PCI e adotta molte idee provenienti dal mondo delle reti locali e in particolare dalla commutazione Ethernet. L'idea base si riduce a questo: in fondo in fondo un PC è un insieme di CPU, memoria e chip controllori di I/O, che necessitano di essere connessi fra loro. Quello che fa PCI Express è fornire un commutatore di uso generale per connettere i chip mediante collegamenti seriali. La Figura 3.57 illustra una tipica configurazione.

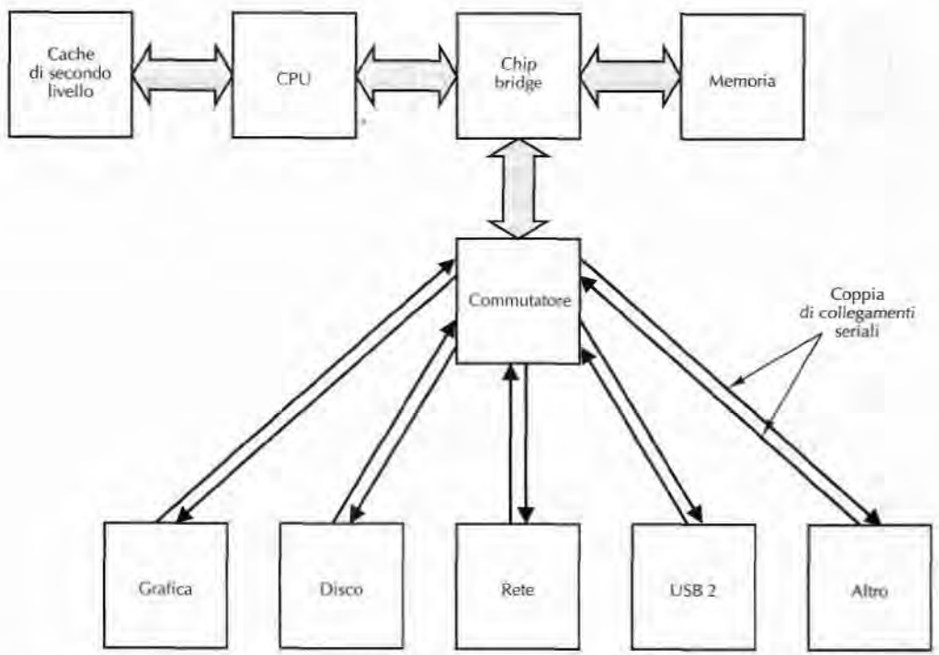


Figura 3.57 Tipico sistema PCI Express.

Come mostra la Figura 3.57, CPU, memoria e cache sono connesse al chip bridge in modo tradizionale. La novità è rappresentata da un commutatore connesso al bridge (oppure integrato direttamente all'interno del chip bridge). Ogni chip di I/O ha una connessione dedicata punto-a-punto con il commutatore, che consiste in una coppia di canali unidirezionali, uno che giunge al commutatore e uno che parte da esso. Ciascun canale è composto da due fili, uno per trasportare il segnale e uno per la terra, in modo da fornire un'elevata immunità al rumore durante le connessioni ad alta velocità. Questa architettura sostituirà quella attuale con uno schema molto più uniforme, nel quale tutti i dispositivi sono trattati in modo uguale.

L'architettura PCI Express differisce da quella vecchia basata sul bus PCI in tre aspetti principali. Due di loro sono già stati esaminati: un commutatore centralizzato al posto di un bus con più connessioni e l'uso di strette connessioni seriali punto-a-punto al posto di un ampio bus parallelo. La terza differenza è più sottile. Il modello concettuale del bus PCI è quello di un master che lancia allo slave un comando per leggere una parola oppure un blocco di parole. Il modello di PCI Express è invece quello di un dispositivo che spedisce un pacchetto di dati a un altro dispositivo. Il concetto di **pacchetto**, che consiste di un'intestazione e di un campo dati, deriva dal mondo delle reti. L'**intestazione** contiene le informazioni di controllo, eliminando quindi la necessità dei vari segnali di controllo presenti sul bus PCI; il **campo dati** contiene invece i dati che devono essere trasferiti. Un PC con PCI Express è in realtà una rete a commutazione di pacchetto in miniatura.

Oltre a queste tre principali rotture con il passato vi sono anche altre differenze. Per esempio, l'utilizzo di un codice per la rilevazione degli errori nei pacchetti fornisce un grado di affidabilità più elevato rispetto a quello di PCI. Inoltre, la connessione tra un chip e il commutatore è diventata più lunga, arrivando fino a 50 cm e permettendo il partizionamento del sistema; si introduce l'espandibilità, dato che un dispositivo può essere in realtà un altro commutatore (fino a un massimo di tre). E ancora: i dispositivi sono inseribili "a caldo", cioè possono essere aggiunti o rimossi dal sistema mentre è in funzione. Infine, dato che le connessioni seriali sono molto più piccole dei vecchi connettori PCI, i dispositivi e i calcolatori possono essere realizzati in dimensioni inferiori. Insomma, PCI Express rappresenta una deviazione molto significativa rispetto alla strada del bus PCI.

Pila di protocolli di PCI Express

Il sistema PCI Express, rimanendo aderente al modello delle reti a commutazione di pacchetto, possiede una pila stratificata di protocolli. Un **protocollo** è un insieme di regole che governano la comunicazione tra due parti. Una pila di protocolli è una gerarchia di protocolli, che gestisce diversi problemi in livelli distinti. Consideriamo per esempio una lettera commerciale. Esistono precise convenzioni riguardo il posizionamento e il contenuto dell'intestazione, l'indirizzo del destinatario, la data, la formula di saluto, il formato, la firma, e così via. Tutto ciò potrebbe essere visto analogamente. Inoltre esiste un altro gruppo di convenzioni riguardo la busta, come la sua dimensione, dove e in che formato indicare gli indirizzi del mittente e del destinatario, dove affrancare il francobollo, e così via. Questi due livelli e i loro protocolli sono indipendenti l'uno dall'altro. È possibile per esempio cambiare completamente l'impaginazione della lettera, pur utilizzando la stessa busta o viceversa. La stratificazione dei protocolli consente una progettazione modulare e flessibile e per decenni è stata usata ampiamente nel mondo del software delle reti. La novità è che in questo caso viene adottata nell'hardware del "bus". La pila di protocolli di PCI Express è mostrata nella Figura 3.58(a).

Esaminiamo i livelli a partire dal basso. Il livello inferiore è il **livello fisico** che tratta lo spostamento dei bit da un mittente a un destinatario lungo una connessione punto-a-punto. Ciascuna connessione punto-a-punto consiste in una o più coppie di collegamenti simplex (cioè unidirezionali). Nel caso più semplice vi è soltanto una coppia in ciascuna direzione, ma se ne possono avere anche 2, 4, 8, 16 o 32. Ciascun collegamento è chiamato **corsia** e il numero di corsie in ciascuna direzione deve essere lo stesso. La prima generazione di prodotti deve supportare un tasso di trasferimento dati in ciascuna direzione di almeno 2,5 Gbps, ma ci si aspetta che la velocità in ciascuna direzione raggiungerà fra non molto i 10 Gbps.

Diversamente dai bus ISA/EISA/PCI, PCI Express non ha un clock principale. I dispositivi sono liberi di trasmettere non appena hanno dei dati da spedire; questa libertà rende più veloce il sistema, ma solleva un problema. Supponiamo che il bit 1 sia codificato con +3 volt e il bit 0 con 0 volt. Se i primi byte sono tutti 0 come può il destinatario sapere che ci sono dei dati in fase di trasmissione? Dopo tutto una sequenza di bit 0 ha lo stesso identico aspetto di un collegamento inattivo. Questo problema è risolto utilizzando quella che viene chiamata **codifica 8b/10b**, in cui si utilizza un simbolo a 10 bit per codificare 1 byte di dati effettivi. Fra tutte le 1024 possibili stringhe di 10 bit sono state opportunamente scelte quelle che hanno sufficienti transizioni del clock per tenere sincronizzati il mittente e il destinatario sul limite di un bit, anche senza l'utilizzo di un clock principale. La conseguenza della codifica 8b/10b è che un link con la capacità grezza di 2,5 Gbps può trasportare soltanto 2 Gbps di dati (netti) dell'utente.

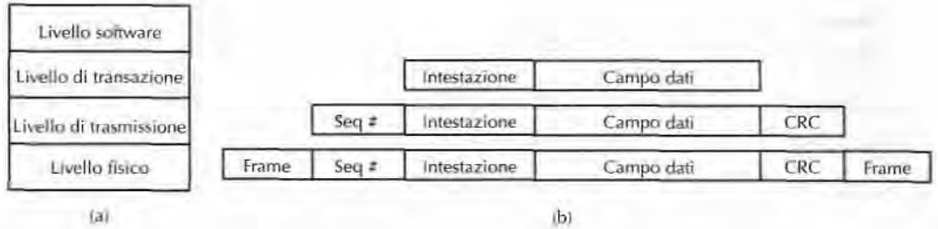


Figura 3.58 (a) Pila di protocolli di PCI Express. (b) Formato di un pacchetto.

Se il livello fisico gestisce la trasmissione di bit, il **livello di trasmissione** si occupa della trasmissione dei pacchetti. Esso prende l'intestazione e il campo dati fornitigli dal livello di transazione e vi aggiunge un numero di sequenza e un codice a correzione di errore chiamato **CRC** (*Cyclic Redundancy Check*, "controllo a ridondanza ciclica"). Il CRC viene generato eseguendo un particolare algoritmo sull'intestazione e sul campo dati. Quando il destinatario riceve un pacchetto esegue la stessa computazione sull'intestazione e sul campo dati e confronta il proprio risultato con il CRC trasportato nel pacchetto. Se corrispondono, restituisce un breve **pacchetto di ackowledgment** ("conferma") per confermare la corretta ricezione del pacchetto. Se invece i due valori non corrispondono, il destinatario richiede una ritrasmissione. In questo modo l'integrità dei dati risulta maggiore rispetto ai sistemi basati sul bus PCI, che non si preoccupano di verificare ed eventualmente ritrasmettere i dati lungo il bus.

Si utilizza inoltre un meccanismo di controllo di flusso per evitare che un mittente veloce sommerga di pacchetti un destinatario più lento. Nel meccanismo utilizzato un destinatario dà al mittente un certo numero di crediti, corrispondenti alla quantità di spazio di buffer che ha a disposizione per memorizzare i pacchetti in entrata. Quando terminano i crediti il mittente deve interrompere la spedizione finché non ottenga nuovi crediti. Questo schema, utilizzato diffusamente in tutte le reti, evita che i dati vadano persi a causa di una differenza di velocità tra il mittente e il destinatario.

Il **livello di transazione** gestisce le azioni sul bus. La lettura di una parola dalla memoria richiede due transazioni: una iniziata dalla CPU o dal canale DMA per richiedere alcuni dati e una iniziata dal mittente per fornire i dati. I compiti del livello di transazione vanno però oltre la semplice gestione di letture e scritture. Esso aggiunge un valore al pacchetto di trasmissione fornitogli dal livello di trasmissione. Per iniziare la trasmissione può dividere ciascuna corsia in più **circuiti virtuali**, fino a un massimo di otto, ciascuno dei quali gestisce una diversa classe del traffico. Il livello di transazione può etichettare i pacchetti in base alla loro classe di traffico, che può comprendere attributi che indicano per esempio: alta priorità, bassa priorità, non effettuare lo snooping, ammettere la consegna in ordine arbitrario, e altro ancora. Il commutatore può utilizzare questi tag quando deve scegliere il nuovo pacchetto da gestire.

Ciascuna transazione utilizza uno dei quattro seguenti spazi d'indirizzi.

1. Spazio di memoria (per letture e scritture ordinarie).
2. Spazio di I/O (per indirizzare i registri del dispositivo).
3. Spazio di configurazione (per l'inizializzazione del sistema, ecc.).
4. Spazio dei messaggi (per la segnalazione, gli interrupt, ecc.).

Lo spazio di memoria e lo spazio di I/O sono simili a quelli dei sistemi attuali. Lo spazio di configurazione può essere utilizzato per implementare funzionalità come il plug-and-play. Lo spazio dei messaggi prende il ruolo di molti dei segnali di controllo esistenti, ed è necessario in quanto nessuna delle linee di controllo del bus PCI è presente in PCI Express.

Il **livello software** che interfaccia il sistema PCI Express al sistema operativo può emulare il bus PCI in modo che sia possibile utilizzare sistemi operativi non modificati. Operando in questo modo ovviamente non si trae pieno vantaggio dalla potenza di PCI Express; la retrocompatibilità è tuttavia necessaria finché i sistemi operativi non saranno modificati per utilizzare appieno PCI Express. L'esperienza mostra che ciò può richiedere un certo periodo di tempo.

La Figura 3.58(b) mostra il flusso d'informazioni. Quando viene passato un comando a livello software, esso lo passa a livello di transazione, che lo riformula in termini di intestazione e campo dati. Queste due parti sono poi passate a livello di trasmissione, che aggiunge un numero di sequenza in testa e il CRC in coda. Questo pacchetto, allungato, viene successivamente passato a livello fisico, che aggiunge a entrambe le estremità informazioni per formare un pacchetto fisico che infine può essere effettivamente trasmesso. Dal lato del destinatario si svolge il processo inverso, durante il quale vengono eliminate l'intestazione e la coda e il risultato viene passato a livello di transazione.

Il concetto secondo il quale ogni livello aggiunge nuova informazione ai dati mentre lo passa al protocollo sottostante è stato utilizzato per decenni e con grande successo nel mondo delle reti. La differenza principale tra una rete e PCI Express è che nel campo delle reti il codice dei vari livelli è quasi sempre una parte software del sistema operativo, mentre con PCI Express fa interamente parte dell'hardware del dispositivo.

PCI Express è un argomento complesso e, per maggiori informazioni, si consultino (Mayhew e Krishnan, 2003; Solari e Congdon, 2005).

3.6.4 Universal Serial Bus

I bus PCI e PCI Express sono adatti per collegare periferiche ad alta velocità al calcolatore, ma sono troppo costosi per dispositivi a bassa velocità come tastiere e mouse. Tradizionalmente i dispositivi standard di I/O erano connessi al calcolatore in un modo specifico, mentre restavano liberi alcuni slot ISA e PCI per aggiungere nuovi dispositivi. Purtroppo questo schema è stato fonte di problemi fin dall'inizio.

Per esempio, ciascun nuovo dispositivo di I/O dispone sia della scheda ISA sia di quella PCI e spesso è compito dell'utente impostare interruttori e ponticelli per assicurarsi che non vadano in conflitto con le altre schede. L'utente deve quindi aprire il contenitore, inserire attentamente la scheda, chiudere il contenitore e riavviare il sistema. Per molti utenti queste operazioni appaiono complesse e si prestano a errori; inoltre il numero di slot ISA e PCI è molto limitato (in genere due o tre). Anche con le schede plug-and-play l'utente deve aprire il calcolatore per inserire la scheda; inoltre non si risolve il problema dei pochi slot a disposizione.

Nel 1993, per risolvere questo problema, i rappresentanti di sette società (Compaq, DEC, IBM, Intel, Microsoft, NEC, e Northern Telecom) si riunirono per progettare un modo migliore per collegare a un calcolatore periferiche a bassa velocità. In seguito, a queste società se ne aggiunsero altre centinaia e, nel 1998 venne rilasciato uno standard detto **USB** (*Universal Serial Bus*, "bus seriale universale") che oggi è implementato diffusamente nei personal computer. Lo standard è descritto in modo approfondito in Anderson (1997) e Tan (1997).

In seguito sono elencati alcuni degli obiettivi delle aziende che hanno concepito originariamente USB e che hanno dato via al progetto.

1. Gli utenti non dovrebbero essere obbligati a impostare interruttori e contatti sulle schede dei dispositivi.
2. Gli utenti non dovrebbero essere obbligati ad aprire il computer per installare nuovi dispositivi di I/O.
3. Dovrebbe esserci un solo tipo di cavo per connettere tutti i dispositivi.
4. I dispositivi di I/O dovrebbero trarre la propria alimentazione dal cavo.
5. Dovrebbe essere possibile collegare fino a 127 dispositivi a un calcolatore.
6. Il sistema dovrebbe supportare dispositivi funzionanti in tempo reale (per esempio dispositivi audio, telefoni).
7. I dispositivi dovrebbero essere installabili a calcolatore acceso.
8. Non dovrebbe essere necessario riavviare il sistema dopo aver installato un nuovo dispositivo.
9. Il nuovo bus e i suoi dispositivi di I/O dovrebbero avere bassi costi di produzione.

USB raggiunge tutti questi obiettivi. USB è progettato per dispositivi a bassa velocità come tastiere, mouse, macchine fotografiche digitali, scanner, telefoni digitali, ecc. La Versione 1.0 ha una larghezza di banda di 1,5 Mbps, che è sufficiente per le tastiere e i mouse, mentre la Versione 1.1 funziona a 12 Mbps, un valore sufficiente per stampanti, macchine fotografiche digitali e molti altri dispositivi. Questi limiti sono stati scelti in modo da contenere i costi.

Un sistema USB è composto da un **hub principale** che si collega al bus del sistema (vedi Figura 3.52) e che possiede delle prese per i cavi che connettono i dispositivi di I/O o per hub di espansione, in modo da fornire un maggior numero di prese. La topologia di un sistema USB è quindi un albero la cui radice è l'hub principale. I cavi hanno connettori diversi sull'estremità dell'hub e su quella del dispositivo, in modo da evitare che l'utente possa accidentalmente connettere due prese dell'hub.

Il cavo consiste in quattro collegamenti: due per i dati, uno per l'alimentazione (+5 volt) e uno per la terra. Il sistema di segnalazione trasmette uno 0 come una transizione di tensione e un 1 come l'assenza di transizione; in questo modo una lunga sequenza di 0 genera un impulso costante.

Quando viene collegato un nuovo dispositivo l'hub principale rileva l'evento e interrompe il sistema operativo; quest'ultimo interroga il dispositivo per sapere di che periferica si tratta e di quanta banda ha bisogno. Se il sistema operativo decide che c'è sufficiente lar-

ghezza di banda per il dispositivo, gli assegna un indirizzo univoco (tra 1 e 127) e scarica nel dispositivo, oltre a questo indirizzo, anche altre informazioni necessarie a configurare i registri. In questo modo è possibile connettere nuovi dispositivi "al volo", senza alcuna configurazione da parte dell'utente e senza dover installare una scheda ISA o PCI. Le schede non inizializzate hanno indirizzo 0 e possono dunque essere indirizzate. Per rendere più semplici i collegamenti molti dispositivi USB contengono al loro interno un hub, in modo da poter accettare ulteriori periferiche USB. Un monitor può per esempio avere due porte USB per potervi collegare gli altoparlanti destro e sinistro.

Da un punto di vista logico il sistema USB può essere visto come un insieme di *pipe* (condotti) di bit che partono dall'hub principale e arrivano alle periferiche di I/O. Ciascun dispositivo può dividere la sua pipe in un massimo di 16 condotti per diversi tipi di dati (per esempio, audio e video). All'interno di ogni condotto i dati viaggiano dall'hub principale alla periferica o viceversa, mentre non esiste alcun traffico di dati tra due dispositivi di I/O.

Esattamente ogni 1,00 ± 0,05 ms, l'hub principale spedisce in broadcast un nuovo frame per mantenere sincronizzati tutti i dispositivi. Un frame è associato a un condotto di bit e consiste in pacchetti, il primo dei quali viaggia dall'hub principale verso il dispositivo. I pacchetti successivi di un frame possono viaggiare nello stesso senso oppure dal dispositivo all'hub principale. La Figura 3.59 mostra una sequenza di quattro frame. Come si può vedere, dato che i frame 0 e 2 non implicano alcuna azione è sufficiente un pacchetto SOF (*Start of Frame*, "inizio del pacchetto"). Questo pacchetto viene sempre spedito in broadcast a tutti i dispositivi. Per fare un esempio il frame 1 potrebbe essere un'interrogazione a uno scanner, affinché restituisca i bit che ha rilevato nell'immagine che sta scannerizzando; il frame 3 potrebbe consistere invece nella spedizione di dati a qualche dispositivo, per esempio una stampante.

USB supporta quattro tipi di frame: controllo, isocrono, *bulk* e interrupt. I frame di controllo sono usati per configurare i dispositivi, assegnare loro dei comandi e interrogarli sul loro stato. I frame isocroni servono per i dispositivi funzionanti in tempo reale (come microfoni, altoparlanti o telefoni) che devono spedire o accettare dati a precisi intervalli temporali. Hanno un ritardo altamente prevedibile, ma non forniscono alcuna ritrasmissione in caso di errore.

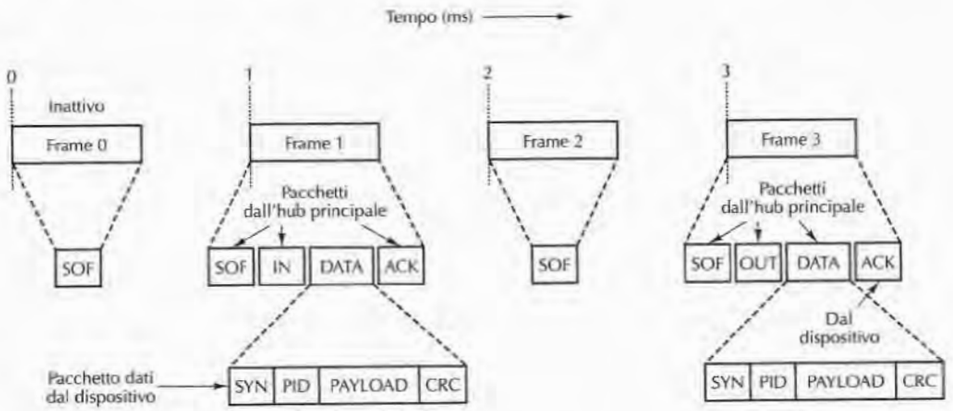


Figura 3.59 L'hub principale USB spedisce frame ogni millisecondo.

I frame bulk sono utilizzati per trasferimenti di grandi dimensioni per dispositivi che non richiedono un funzionamento in tempo reale, come le stampanti. Infine i frame di interrupt sono necessari in quanto USB non supporta gli interrupt. Per esempio, invece di avere una tastiera che provoca un interrupt ogni volta che viene premuto un tasto, il sistema operativo la interroga ogni 50 ms per raccogliere i dati pendenti relativi alla pressione dei tasti.

Un frame è composto da uno o più pacchetti, alcuni dei quali possono spostarsi in entrambe le direzioni. Esistono quattro tipi di pacchetti: *token*, dati, *handshake* e pacchetti speciali. I pacchetti token (come SOF, IN e OUT della Figura 3.59) viaggiano dall'hub principale fino alla periferica e servono per il controllo del sistema. Il pacchetto SOF (*Start of Frame*) è il primo di ogni frame e ne marca l'inizio: se non c'è alcuna operazione da compiere esso è l'unico pacchetto del frame. Il pacchetto token IN è un'interrogazione (*poll*) utilizzata per richiedere al dispositivo la resistuzione di alcuni dati. I suoi campi indicano quale flusso di bit viene interrogato in quel momento, in modo che il dispositivo possa sapere quale dato deve restituire (nel caso in cui abbia più flussi). Il pacchetto token OUT segnala che seguiranno dei dati per il dispositivo. Un quarto tipo di pacchetto token, SETUP (non mostrato nella figura), è utilizzato invece per la configurazione.

Il formato dei pacchetto dati (mostrato nella Figura 3.59) consiste in un campo di 8 bit per la sincronizzazione, in un identificatore a 8 bit del tipo di pacchetto (PID), nel campo dati e in un CRC a 16 bit per rilevare gli errori. Esistono tre tipi di pacchetti handshake: ACK (il pacchetto precedente è stato ricevuto correttamente), NAK (è stato rilevato un errore CRC) e STALL (attendere prego, in questo momento sono occupato).

Relativamente alla Figura 3.59, osserviamo che ogni millisecondo l'hub principale deve spedire un frame, anche se non deve essere effettuata alcuna operazione. I frame 0 e 2 consistono semplicemente di un pacchetto SOF, che indica che non vi è nulla da fare. Il frame 1 è un'interrogazione e inizia quindi con i pacchetti SOF e IN che viaggiano dal calcolatore alla periferica, seguiti poi da un pacchetto DATA dalla periferica al calcolatore. Il pacchetto ACK comunica alla periferica che i dati sono stati ricevuti correttamente; in caso di errore viene invece restituito un pacchetto NAK e viene rispedito il pacchetto bulk di dati (ma ciò non avviene per i dati isocroni). La struttura del frame 3 è simile a quella del frame 1, tranne per il fatto che in questo caso il flusso di dati va dal calcolatore al dispositivo.

Dopo aver terminato, nel 1998, la definizione dello standard USB, i tecnici coinvolti nel progetto, non avendo nient'altro da fare, cominciarono a lavorare a una nuova versione ad alta velocità di USB: **USB 2.0**. Questo standard è simile al vecchio USB 1.1 ed è retrocompatibile con esso, tranne per il fatto che aggiunge una terza velocità, pari a 480 Mbps, alle due già esistenti. Sono state inoltre introdotte alcune differenze di minor entità, che riguardano per esempio l'interfaccia tra hub principale e controllore. USB 1.1 ammetteva due tipi d'interfaccia. Il primo, **UHCI** (*Universal Host Controller Interface*), fu progettato da Intel lasciando gran parte dell'onere agli ingegneri software (leggi: Microsoft). Il secondo, **OHCI** (*Open Host Controller Interface*), fu progettato da Microsoft scaricando gran parte del lavoro agli ingegneri hardware (leggi: Intel). In USB 2.0 tutti si sono trovati d'accordo nell'utilizzare una nuova e unica interfaccia chiamata **EHCI** (*Enhanced Host Controller Interface*).

USB, che ora funziona a 480 Mbps, è diventato un chiaro concorrente del bus seriale IEEE 1394, indicato di solito con il nome FireWire, che raggiunge la velocità di 400 Mbps. Anche se praticamente qualsiasi nuovo Pentium è dotato di USB 2.0, lo standard 1394 non scomparirà; esso ha infatti il sostegno del mercato dell'elettronica di consumo. Nel prossimo

futuro videocamere digitali, lettori DVD e simili dispositivi continueranno a essere equipaggiati con interfacce 1394, dato che i loro produttori non vogliono affrontare la spesa necessaria per passare a un nuovo standard, anche se è decisamente migliore di quello adottato. Anche l'utente finale, tuttavia, non ama cambiare gli standard.

3.7 Interfacce

Un classico calcolatore di piccole e medie dimensioni è composto da un chip della CPU, da alcuni chip di memoria e da alcuni controllori di I/O, tutti connessi fra loro mediante un bus. Abbiamo già esaminato alcuni dettagli delle memorie, delle CPU e dei bus. È giunto ora il momento di analizzare l'ultima tessera del puzzle, i chip di I/O; è attraverso questi chip che il calcolatore comunica con il mondo esterno.

3.7.1 Chip di I/O

Numerosi chip di I/O sono già disponibili e con il passare del tempo ne vengono introdotti sempre di nuovi. Chip molto comuni sono gli UART, gli USART, i controllori dei CRT, i controllori dei dischi e i PIO. Un chip **UART** (*Universal Asynchronous Receiver Transmitter*) è un chip che può leggere un byte da un bus di dati e generarlo in output, un bit alla volta, su una linea seriale per un terminale, oppure può ricevere input da un terminale. Generalmente i chip UART consentono velocità comprese tra 50 e 19.200 bps, dimensione dei caratteri variabili tra 5 e 8 bit, uno, uno e mezzo o due bit di stop e possono eventualmente fornire un controllo sulla parità pari o dispari; tutto questo avviene sotto il controllo del software. I chip **USART** (*Universal Synchronous Asynchronous Receiver Transmitters*) possono gestire trasmissioni sincrone utilizzando vari protocolli, oltre a supportare tutte le funzionalità dei chip UART. Dato che nel corso del Capitolo 2 abbiamo già visto i chip UART, analizzeremo ora l'interfaccia parallela come esempio di chip di I/O.

Chip PIO

Un esempio tipico di chip è il **PIO** (*Parallel Input/Output*) Intel 8255A. Questo chip (mostrato nella Figura 3.60) ha 24 linee di I/O che possono essere interfacciate a qualsiasi dispositivo compatibile con TTL, come tastiere, interruttori, luci o stampanti. In poche parole il programma della CPU può scrivere uno 0 o un 1 su ogni linea, oppure può leggere

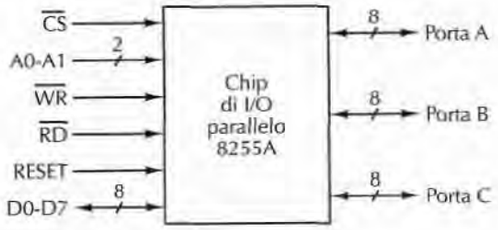


Figura 3.60 Chip PIO 8255A.

in input lo stato di ogni linea, garantendo così un'elevata flessibilità. Un piccolo sistema basato su CPU che usa PIO può spesso sostituire, specialmente nei sistemi integrati, un'intera scheda piena di chip SSI o MSI.

Anche se la CPU può configurare l'8255A in modi diversi operando sullo stato dei registri all'interno del chip, ci concentreremo soltanto su alcune delle sue modalità di funzionamento. Il modo più semplice di usare l'8255A è con tre porte indipendenti a 8 bit, A, B e C, a ciascuna delle quali è associato un registro latch a 8 bit. Per impostare le linee su una porta, la CPU scrive semplicemente un numero a 8 bit nel registro corrispondente e tale numero rimane sulla linea di output finché il registro non venga riscritto. Per utilizzare una porta in input, la CPU non fa che leggere il registro corrispondente.

Altre modalità di funzionamento prevedono l'handshaking con dispositivi esterni. Per esempio, per spedire un output verso un dispositivo che non è sempre pronto ad accettare i dati, l'8255A può renderli disponibili su una porta di output e attendere che il dispositivo restituisca un segnale per indicare che ha accettato i dati e che ne desidera degli altri. La logica necessaria per memorizzare questi impulsi e renderli disponibili alla CPU è integrata nell'hardware dell'8255A.

Dal diagramma funzionale dell'8255A possiamo vedere che oltre ai 24 pin necessari per le tre porte esso dispone di otto linee che lo connettono direttamente al bus dei dati, una linea per la selezione del chip, le linee per la lettura e la scrittura, due linee d'indirizzo e una linea per resettare il chip. Le due linee d'indirizzo selezionano uno dei quattro registri interni, corrispondenti alle porte A, B, C e al registro di stato, i cui bit, tra l'altro, determinano quali porte sono utilizzate per l'input e quali per l'output. Di solito le due linee d'indirizzo sono connesse ai bit meno significativi del bus degli indirizzi.

3.7.2 Decodifica dell'indirizzo

Finora siamo stati intenzionalmente vaghi circa il modo in cui la selezione del chip è asserita sulla memoria e sui chip di I/O. Per vedere con maggior attenzione come ciò si svolge, consideriamo un semplice calcolatore integrato a 16 bit composto da una CPU, una EPROM per il programma da 2KB × 8 byte, una RAM per i dati da 2KB × 8 byte e un chip PIO. Questo piccolo sistema potrebbe essere utilizzato come prototipo del cervello di un giocattolo economico o di un semplice elettrodomestico. Una volta entrata in fase produttiva la EPROM dovrebbe essere sostituita da una ROM.

Il chip PIO può essere selezionato come un vero dispositivo di I/O o come parte della memoria. Se scegliamo di usarlo come un dispositivo di I/O, dobbiamo selezionarlo usando una linea di bus che indica esplicitamente che ci si sta riferendo a un dispositivo di I/O e non a una memoria. Se scegliamo il secondo approccio, quello dell'**I/O mappato in memoria**, dobbiamo assegnargli 4 byte dello spazio della memoria, necessari per indirizzare le tre porte e il registro di controllo. La scelta è praticamente arbitraria; noi sceglieremo l'I/O mappato in memoria in quanto mette in evidenza alcuni interessanti problemi riguardanti l'interfaccia di I/O.

Sia la EPROM sia la RAM richiedono uno spazio degli indirizzi di 2 KB, mentre il chip PIO richiede soltanto 4 byte. Dato che lo spazio degli indirizzi del nostro esempio è di 64 KB dobbiamo semplicemente scegliere dove collocare i tre dispositivi. La Figura 3.61 mostra una scelta possibile. La EPROM occupa gli indirizzi fino a 2 KB, la RAM occupa gli indirizzi compresi tra 32 e 34 KB, e il chip PIO occupa gli ultimi 4 byte dello spazio degli indirizzi, da

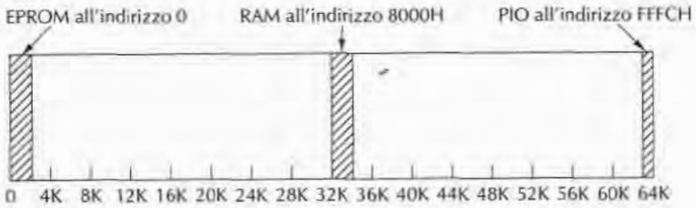


Figura 3.61 Posizione della EPROM, della RAM e del chip PIO nel nostro spazio d'indirizzamento di 64 KB.

65532 a 65535. Dal punto di vista del programmatore lo spazio degli indirizzi scelto non fa alcuna differenza; dal punto di vista dell'interfaccia invece sì. Se avessimo scelto d'indirizzare il chip PIO mediante lo spazio di I/O non ci sarebbe stato bisogno di alcun indirizzo di memoria (ma sarebbero stati necessari quattro spazi degli indirizzi di I/O).

Usando l'assegnamento degli indirizzi della Figura 3.61 la EPROM potrebbe essere selezionata da uno qualsiasi degli indirizzi di memoria a 16 bit della forma 00000xxxxxxxxxxx (binario). In altre parole ogni indirizzo in cui i 5 bit più significativi sono 0 cade nei 2 KB bassi della memoria, cioè nella EPROM. La selezione del chip della EPROM potrebbe quindi essere collegata a un comparatore a 5 bit con input impostato al valore 00000.

Un modo migliore per ottenere lo stesso risultato consiste nell'usare una porta `OR` a cinque ingressi collegati alle linee d'indirizzo che vanno da `A11` a `A15`. Se e solo se tutte le linee valgono 0 l'output varrà 0, asserendo di conseguenza il segnale $\overline{CS}$ (asserito con il valore basso). Sfortunatamente non esiste nessuna porta `OR` a cinque ingressi nella serie SSI standard; quella più simile di cui possiamo disporre è una porta `NOR` a otto ingressi. Collegando tre input con la terra e invertendo l'output possiamo comunque generare il segnale desiderato, come mostra la Figura 3.61. I chip SSI sono talmente economici che, tranne in particolari circostanze, non è un problema utilizzarne uno in modo non del tutto efficiente. Per convenzione nei diagrammi dei circuiti non vengono mostrate le porte inutilizzate.

Si può utilizzare lo stesso principio anche per la RAM, che però dovrebbe rispondere agli indirizzi binari della forma 10000xxxxxxxxxxx. Quindi, come mostra la figura, è necessario un invertitore aggiuntivo. La decodifica degli indirizzi del chip PIO è invece leggermente più complicata, dato che il chip viene selezionato dai quattro indirizzi della forma 11111111111111xx. Nella figura è mostrato un circuito che asserisce $\overline{CS}$ soltanto quando sul bus degli indirizzi appare l'indirizzo corretto. Il circuito utilizza due porte `NAND` per alimentare una porta `OR`. Utilizzando SSI per costruire la logica di decodifica degli indirizzi della Figura 3.62(a) sono necessari sei chip: i quattro chip a otto input, una porta `OR` e un chip con tre invertitori.

Tuttavia, se il calcolatore è realmente composto soltanto dalla CPU, da due chip di memoria e dal chip PIO, possiamo ricorrere a un trucco che semplifica di gran lunga la decodifica degli indirizzi. Il trucco è basato sul fatto che tutti e soli gli indirizzi della EPROM hanno uno 0 nel bit più significativo, `A15`. Quindi, come mostra la Figura 3.62(b), possiamo semplicemente collegare $\overline{CS}$ direttamente ad `A15`.

A questo punto la scelta di collocare la RAM a 8000H appare meno arbitraria. La decodifica della RAM può basarsi sull'osservazione che gli unici indirizzi validi della forma

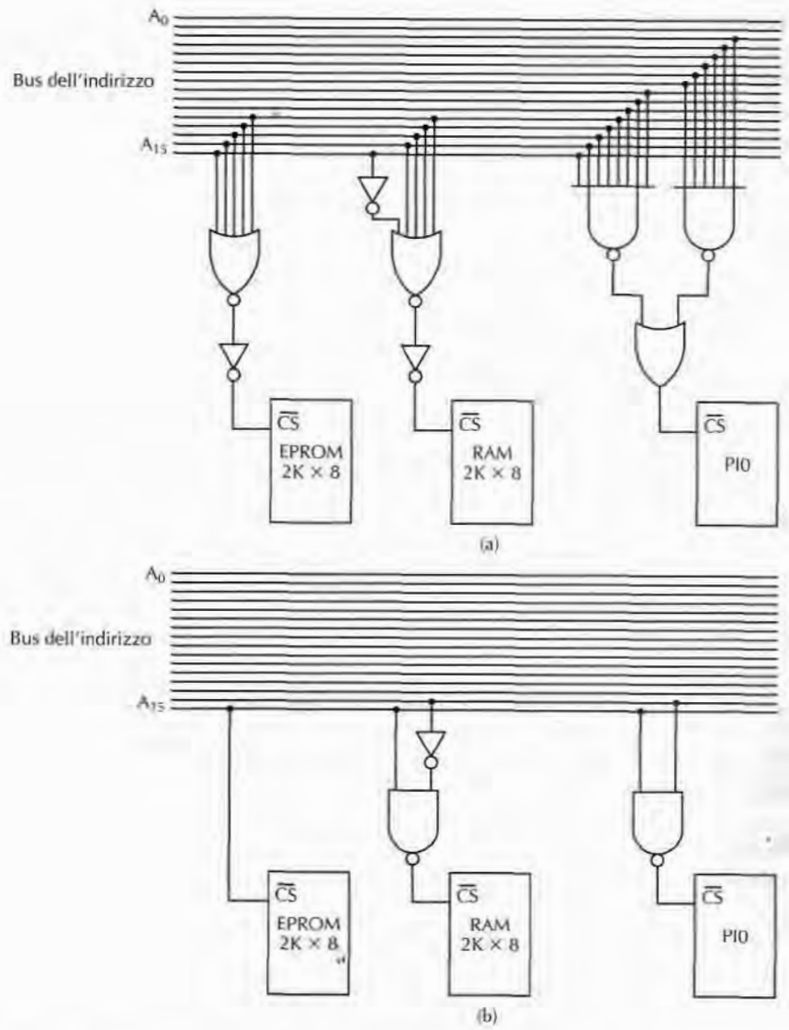


Figura 3.62 Decodifica dell'indirizzo: (a) completa; (b) parziale.

10xxxxxxxxxxxxxx ricadono nella RAM; è quindi sufficiente decodificare 2 bit. In modo analogo ogni indirizzo che inizia con 11 deve per forza essere un indirizzo del chip PIO. La logica per la decodifica completa può quindi essere costituita solamente da due porte `NAND` e da un invertitore. Dato che un invertitore è realizzabile collegando i due ingressi di una porta `NAND`, tutto ciò di cui abbiamo bisogno è un chip con quattro porte `NAND`.

La logica di decodifica degli indirizzi della Figura 3.62(b) è chiamata **decodifica parziale dell'indirizzo**, dato che non utilizza l'indirizzo completo. Una proprietà di

questo metodo è che la lettura degli indirizzi 0001000000000000, 0001100000000000 o 0010000000000000 fornirà lo stesso risultato. In realtà qualsiasi indirizzo della metà inferiore della memoria selezionerà la EPROM; ciò non provocherà alcun danno dato che gli indirizzi extra non saranno utilizzati. Se però si sta progettando un calcolatore che potrà in futuro essere espanso (cosa poco probabile nel caso di un giocattolo) bisognerebbe evitare la decodifica parziale, poiché vincola eccessivamente lo spazio degli indirizzi.

Un'altra comune tecnica d'indirizzamento consiste nell'usare un decodificatore, come quello mostrato nella Figura 3.13. Connettendo i tre input alle tre linee dell'indirizzo più significative otteniamo otto output, corrispondenti agli indirizzi nei primi 8KB, a quelli nei secondi 8KB, e così via. Per un calcolatore con otto RAM di 8KB × 8, un simile chip fornisce una decodifica completa. Anche per un calcolatore con otto chip di memoria 2KB × 8 è sufficiente un singolo decodificatore, fermo restando che i chip di memoria devono essere collocati in diversi settori da 8 KB dello spazio degli indirizzi. Ricordiamoci della precedente osservazione su quanto sia importante la posizione dei chip della memoria e dei chip di I/O all'interno dello spazio degli indirizzi.

3.8 Riepilogo

I calcolatori sono costruiti a partire da chip di circuiti integrati contenenti piccoli elementi di commutazione chiamati porte logiche, tra cui le più comuni sono AND, OR, NAND, NOR e NOT. È possibile costruire semplici circuiti combinando direttamente le singole porte.

Alcuni esempi di circuiti sono i multiplexer, i demultiplexer, i codificatori, i decodificatori, gli shifter e le ALU. Utilizzando un PLA è possibile programmare funzioni booleane arbitrarie; se sono necessarie molte funzioni booleane spesso i PLA risultano ancora più efficienti. Utilizzando le leggi dell'algebra di Boole è possibile trasformare un circuito in un circuito equivalente dalla forma diversa; in molti casi usando queste tecniche è possibile produrre circuiti più economici.

L'aritmetica dei calcolatori è realizzata mediante sommatori. Per sommare una coppia di bit si possono usare due semisommatori. Un sommatore per una parola composta da più bit può essere realizzato connettendo più sommatori in cascata.

I componenti delle memorie (statiche) sono i latch e i flip-flop, ciascuno dei quali può memorizzare un bit d'informazione. Possono essere combinati linearmente in latch e flip-flop ottali oppure logaritmicamente in memorie molto più capaci, orientate alla parola. Le memorie possono essere di vario tipo: RAM, ROM, PROM, EPROM, EEPROM e flash. Le memorie statiche non devono essere riaggiornate, dato che mantengono i valori memorizzati fintanto che ricevono alimentazione. Le memorie dinamiche, al contrario, devono essere aggiornate periodicamente per compensare la scarica dei piccoli condensatori che si trovano sul chip.

I componenti di un sistema di un calcolatore sono connessi fra loro mediante bus. Molti pin di un classico chip della CPU guidano direttamente una linea del bus. Le linee del bus possono essere divise in linee d'indirizzo, di dati e di controllo. I bus sincroni sono guidati da un clock principale, mentre i bus asincroni usano una strategia di handshaking per sincronizzare lo slave con il master.

Il Pentium 4 è un esempio di moderna CPU. I sistemi che lo utilizzano hanno un bus di memoria, un bus PCI, un bus ISA e un bus USB. Il bus PCI può trasferire 64 bit alla volta a una frequenza di 66 MHz, il che è sufficiente per quasi tutte le periferiche, ma non per la memoria.

Interruttori, luci, stampanti e molti altri dispositivi di I/O possono essere interfacciati utilizzando chip di I/O parallelo come l'8255A. Secondo le necessità questi chip possono essere configurati per essere parte dello spazio di I/O o parte dello spazio di memoria. A seconda dell'applicazione possono essere decodificati parzialmente oppure in modo completo.

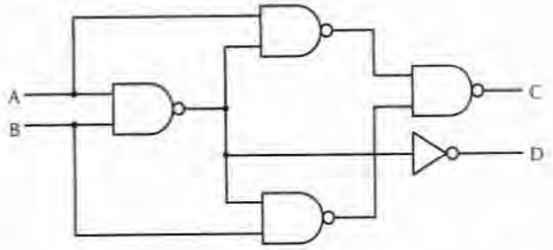
Problemi

1. Un professore di logica matematica entra in un bar e dice: "Voglio un cappuccino oppure un toast e una birra". Sfortunatamente il barista, essendo stato bocciato in quinta elementare, non sa (o non se ne preoccupa) se "e" abbia oppure no la precedenza su "o" ma, per quanto gliene possa interessare, un'interpretazione vale l'altra. Quali dei seguenti casi sono un'interpretazione corretta dell'ordine (si noti che qui la parola "oppure" significa "or esclusivo").
   a. Solo un cappuccino.
   b. Solo un toast.
   c. Solo una birra.
   d. Un toast e una birra.
   e. Un cappuccino e una birra.
   f. Un toast e un cappuccino.
   g. Tutti e tre.
   h. Nulla – il professore non verrà servito visto che si è comportato da sapientone.
2. Un missionario che si è perso nella California del Sud si ferma a un bivio della strada. Sa che nella zona ci sono due bande di motociclisti, una delle quali dice sempre la verità, mentre l'altra mente sempre. Il missionario vuole sapere quale strada porta a Disneyland; quale domanda dovrebbe porre?
3. Si utilizzi una tabella di verità per mostrare la funzione $X = (X \text{ AND } Y) \text{ OR } (X \text{ AND NOT } Y)$.
4. Esistono quattro funzioni booleane di una singola variabile e 16 di due variabili. Quante funzioni di tre variabili esistono? E di n variabili?
5. Si mostri com'è possibile costruire la funzione AND mediante due porte NAND.
6. Utilizzando il multiplexer a tre vie della Figura 3.12 si implementi una funzione il cui output sia la parità degli input; l'output deve quindi essere 1 se e solo se un numero pari degli input vale 1.
7. Proviamo a ragionare; il multiplexer della Figura 3.12 è in grado di calcolare effettivamente un'arbitraria funzione booleana a *quattro* variabili. Si dimostri in che modo, si fornisca un esempio e si disegni il diagramma logico per la funzione che vale 0 se il numero della riga della tabella di verità si scrive con un numero pari di lettere, mentre vale 1 se le lettere sono dispari. Per esempio, 0000 = zero = 4 lettere → 0; 0111 = sette = 5 lettere → 1; 1101 = tredici = 7 lettere → 1. *Suggerimento*: se chiamiamo D la quarta variabile di input possiamo collegare le otto linee di input a V_{cc}, alla terra, a D oppure a $\overline{D}$.
8. Si disegni il diagramma logico di un codificatore a 2 bit; il circuito ha quattro linee di input di cui una soltanto ha valore alto in un dato istante e due linee di output il cui valore binario a 2 bit indica quale input è alto.

9. Si disegni il diagramma logico di un demultiplexer a 2 bit; il circuito ha una singola linea di input che è indirizzata verso una delle quattro linee di output in base allo stato delle due linee di controllo.

10. Si ridisegni, con sufficiente dettaglio, il PLA delle Figura 3.15, per mostrare com'è possibile implementare la maggior parte delle funzioni logiche della Figura 3.3. In particolare si presti attenzione a mostrare quali connessioni sono presenti in entrambe le matrici.

11. Che cosa fa questo circuito?



12. Un chip MSI molto comune è il sommatore a 4 bit. È possibile agganciare quattro di questi chip per formare un sommatore a 16 bit; quanti pin ci si aspetta che avrà il sommatore? Perché?

13. Un sommatore a n bit può essere costruito collegando in cascata n sommatori in serie, in cui il riporto C_i dello stadio i giunge dall'output dello stadio $i-1$. Il riporto C_0 nello stadio 0 vale 0. Dato che ciascuno stadio richiede T ns per produrre la propria somma e il proprio riporto, il riporto allo stadio i sarà valido solo iT ns dopo l'inizio della somma. Per valori di n elevati il tempo necessario affinché il riporto giunga fino all'ultimo stadio potrebbe essere eccessivamente lungo. Si progetti un sommatore che funzioni più velocemente. *Consiglio*: ciascun C_i può essere espresso in termini dei bit dell'operando A_{i-1} e B_{i-1} e del riporto C_{i-1}. Utilizzando questa relazione è possibile esprimere C_i come una funzione degli input agli stadi compresi tra 0 e $i-1$; è quindi possibile generare simultaneamente tutti i riporti.

14. Si assuma che nella Figura 3.19 tutte le porte logiche abbiano un ritardo di propagazione di 1 ns e che sia possibile ignorare tutti gli altri ritardi. Qual è il tempo minimo al quale si ha la certezza che un circuito progettato in questo modo genererà un output valido?

15. La ALU della Figura 3.20 è in grado di eseguire somme a 8 bit in complemento a due. È in grado di eseguire anche le sottrazioni in complemento a due? In caso affermativo si spieghi in che modo. In caso negativo si modifichi il circuito in modo che sia in grado di compiere le sottrazioni.

16. Una ALU a 16 bit è costruita mediante 16 ALU a 1 bit, ciascuna delle quali richiede 10 ns per compiere una somma. Se è presente un ulteriore ritardo di 1 ns dovuto alla propagazione tra una ALU e la successiva, dopo quanto tempo appare il risultato di una somma a 16 bit?

17. Talvolta per una ALU a 8 bit come quella della Figura 3.20 è utile generare in output il valore costante –1. Si forniscano due modi diversi per raggiungere questo scopo. Per ciascun modo si specifichino i valori dei sei segnali di controllo.

18. Che cos'è lo stato di riposo degli input R e S di un latch SR costruito mediante due porte NAND?

19. Il circuito della Figura 3.26 è un flip-flop pilotato dal fronte di salita del clock. Si modifichi questo circuito per ottenere un flip-flop che sia pilotato dal fronte di discesa del clock.

20. La memoria 4 × 3 della Figura 3.29 utilizza 22 porte AND e tre porte OR. Quante di queste porte sarebbero necessarie se si espandesse il circuito strutturandolo come 256 × 8?

21. Per poter pagare il tuo nuovo personal computer hai deciso di dare consulenze a produttori di chip SSI con poca esperienza. Su richiesta di un'importante cliente uno di questi produttori sta pensando di costruire un chip composto da quattro flip-flop D, ciascuno contenente sia Q sia $\overline{Q}$. Il progetto proposto prevede inoltre che tutti e quattro i segnali di clock siano legati fra loro, mentre non sono presenti né i segnali di preset né di clear. Si dia un parere professionale sul progetto.

22. Man mano che in un singolo chip la dimensione della memoria viene ridotta, aumenta il numero di pin richiesti per indirizzarla. Spesso non è conveniente avere un grande numero di pin dell'indirizzo su un singolo chip. Si pensi a un metodo per indirizzare 2^n parole di memoria utilizzando meno di n pin.

23. Un calcolatore con un bus di dati largo 32 bit utilizza una RAM dinamica implementata con un chip 1Mbit × 1. Qual è la più piccola memoria (in byte) che questo calcolatore può avere?

24. Facendo riferimento al diagramma di temporizzazione della Figura 3.38 si supponga di rallentare il clock da 10 ns (com'è mostrato nella figura) a 20 ns, senza modificare i vincoli di temporizzazione. Nel caso peggiore quanto tempo impiega la memoria per prelevare i dati dal bus durante T_3 dopo che $\overline{\text{MREQ}}$ è stato asserito?

25. Facendo nuovamente riferimento alla Figura 3.38 si supponga di lasciare il clock a 100 MHz, ma di aumentare T_{DS} fino a 4 ns. Sarebbe possibile utilizzare chip di memoria da 10 ns?

26. Nella Figura 3.38(b) si specifica che T_{ML} deve essere di almeno 3 ns. È possibile immaginare un chip in cui ciò è negativo? In particolare, la CPU potrebbe asserire $\overline{\text{MREQ}}$ prima che l'indirizzo diventi stabile? Se sì, perché? Se no, perché?

27. Si assuma che il trasferimento a blocchi della Figura 3.42 venga effettuato sul bus della Figura 3.38. Nel caso di lunghi blocchi di dati quanta larghezza di banda si guadagna usando un trasferimento a blocchi invece che più trasferimenti singoli? Si assuma successivamente che il bus abbia una larghezza di 32 bit invece che di 8 bit e si risponda nuovamente alla domanda.

28. Si indichino con T_{A1} e T_{A2} i tempi di transizione delle linee d'indirizzo della Figura 3.39, con T_{MREQ1} e T_{MREQ2} i tempi di transizione di $\overline{\text{MREQ}}$, e così via. Si scrivano tutte le disuguaglianze determinate dal protocollo di full handshaking.

29. La maggior parte dei bus a 32 bit permette letture e scritture a 16 bit. Esiste qualche ambiguità sulla posizione in cui vengono posti i dati? Si argomenti la risposta.

30. Molte CPU hanno un tipo speciale di ciclo di bus per la conferma degli interrupt. Perché?

31. Un calcolatore a 64 bit con un bus a 200 MHz richiede quattro cicli per leggere una parola a 64 bit. Quanta larghezza di banda consuma la memoria nel caso peggiore?

32. Una CPU a 32 bit con linee d'indirizzo A2–A31 richiede che tutti i riferimenti alla memoria siano allineati; le parole devono quindi essere indirizzate a multipli di 4 byte e le mezze-parole devono essere indirizzate in corrispondenza dei byte pari. I byte possono essere ovunque. Quante sono le combinazioni legali per le letture della memoria e quanti pin sono necessari per esprimerle? Si diano due risposte e si presenti un esempio per ciascuna.

33. Perché un Pentium 4 non può funzionare con un bus PCI a 32 bit senza perdere funzionalità? Dopo tutto altri calcolatori con un bus di dati a 64 bit possono effettuare trasferimenti larghi 32, 16 o anche 8 bit.

34. Si supponga che una CPU abbia una cache di primo livello e una di secondo livello, con tempi di accesso rispettivamente di 1 ns e 2 ns. Il tempo di accesso alla memoria centrale è di 10 ns. Se hanno successo il 20% degli accessi alla cache di primo livello e il 60% degli accessi alla cache di secondo livello, qual è il tempo medio di accesso?

35. È probabile che un sistema integrato basato sull'8051 includa un chip 8255A?

36. Si calcoli la larghezza di banda necessaria per visualizzare un filmato VGA (640 × 480) a 30 fotogrammi al secondo e in true-color. Si assuma che i dati debbano passare sul bus due volte, la prima dal CD-ROM alla memoria e la seconda dalla memoria allo schermo.

37. Quali segnali del Pentium 4 pilotano la linea FRAME# del bus PCI?

38. Quali dei segnali della Figura 3.56 non sono strettamente necessari affinché il protocollo del bus funzioni?

39. Un sistema basato su PCI Express ha collegamenti di 5 Mbps (capacità grezza). Per funzionare a 8x quanti collegamenti sono necessari in ciascuna direzione per i segnali?

40. Tutte le istruzioni di un calcolatore richiedono due cicli di bus, uno per prelevare l'istruzione e uno per prelevare i dati. Ciascun ciclo di bus richiede 10 ns e ciascuna istruzione 20 ns (cioè l'elaborazione interna è trascurabile). Il calcolatore ha un disco con 2048 settori, 512 byte per traccia e una rotazione di 5 ms. Rispetto alla normalità a quale percentuale si riduce la velocità del calcolatore durante un trasferimento DMA a 32 bit, considerando che ciascuno richiede un ciclo di bus?

41. Su un bus USB la massima dimensione del campo dati di un pacchetto dati isocrono è di 1023 byte. Assumendo che un dispositivo possa spedire solo un pacchetto dati per frame, qual è la massima larghezza di banda per un singolo dispositivo isocrono?

42. Nella Figura 3.62(b) si immagini di aggiungere una terza linea di input alla porta NAND che seleziona il chip PIO. Che cosa si otterrebbe se questa linea venisse connessa ad A13?

43. Si scriva un programma per simulare il comportamento di un array $m \times n$ di porte NAND a due input. Questo circuito, contenuto in un solo chip, ha j pin di input e k pin di output. I valori di j, k, m e n sono parametri della simulazione determinati a tempo di compilazione. Il programma dovrebbe iniziare leggendo una "lista dei collegamenti", in cui per ciascun collegamento sono specificati l'input e l'output. Un input può essere uno dei j pin di input oppure l'output di una porta NAND. Un output è uno dei k pin di output oppure un input di una porta NAND. Gli input non utilizzati hanno valore logico 1. Il programma, dopo aver letto la "lista dei collegamenti", dovrebbe stampare l'output di ciascuno dei 2^j possibili input. Chip come questo, composti da un array di porte, sono usati in modo diffuso per definire circuiti personalizzati su un chip; il motivo è che la maggior parte del lavoro (il collocamento dell'array di porte sul chip) è indipendente dal circuito da implementare. Da progetto a progetto cambiano soltanto i collegamenti.

44. Si scriva un programma per leggere due espressioni booleane arbitrarie e vedere se rappresentano la stessa funzione. Il linguaggio di input comprende singole lettere per le variabili booleane, gli operandi AND, OR e NOT e le parentesi. Ciascuna espressione deve essere fornita su una sola linea di input. Il programma deve calcolare le tabelle di verità di entrambe le funzioni e confrontarle.

45. Si scriva un programma per leggere un gruppo di espressioni booleane e calcolare le matrici 24 × 50 e 50 × 6 necessarie per implementarle per mezzo del PLA della Figura 3.15. Il linguaggio di input deve essere lo stesso dell'esempio precedente. Si stampi la matrice sulla stampante in linea.

Capitolo 4

Livello di microarchitettura

Al di sopra del livello logico digitale si trova il livello di microarchitettura. Com'è illustrato nella Figura 1.2 il suo compito consiste nell'implementare il livello ISA (*Instruction Set Architecture*, "architettura dell'insieme d'istruzioni"). Il modo in cui viene progettato il livello di microarchitettura non dipende solamente dall'ISA che si intende implementare, ma anche dagli obiettivi di costi e prestazioni del calcolatore. Molti ISA moderni, in particolare nelle architetture RISC, sono costituiti da istruzioni semplici che generalmente è possibile eseguire in un unico ciclo di clock. Nel caso di ISA più complessi, come quelli del Pentium 4, l'esecuzione di una singola istruzione può invece richiedere più cicli. Per eseguire un'istruzione può essere necessario localizzare gli operandi all'interno della memoria, leggerli e infine memorizzare il risultato nuovamente in memoria. Spesso l'ordinamento di queste operazioni all'interno di una singola istruzione porta a una strategia di controllo diversa rispetto a quella degli ISA più semplici.

4.1 Esempio di microarchitettura

Sarebbe preferibile introdurre gli argomenti del capitolo spiegando i principi generali su cui si basa la progettazione della microarchitettura; sfortunatamente però non esistono principi generali: ogni microarchitettura rappresenta infatti un caso particolare. Per questo motivo analizzeremo in modo dettagliato un esempio pratico. Come avevamo promesso nel Capitolo 1 abbiamo scelto come ISA di esempio un sottoinsieme della Java Virtual Machine e, dato che questo sottoinsieme contiene solo istruzioni su interi, abbiamo deciso di chiamarlo **IJVM**. Nel capitolo successivo tratteremo l'intera JVM.

Inizieremo descrivendo la microarchitettura sopra la quale implementeremo IJVM. Come abbiamo già visto nel corso del Capitolo 1, molte delle architetture che, come IJVM, contengono istruzioni complesse sono implementate mediante la microprogrammazione. IJVM, pur essendo un insieme di piccole dimensioni, rappresenta tuttavia un buon punto di partenza per descrivere il controllo e l'ordinamento delle istruzioni.

La nostra microarchitettura conterrà un microprogramma (registrato in una ROM) il cui compito sarà quello di prelevare, decodificare ed eseguire le istruzioni IJVM. Dato che abbiamo bisogno di un piccolo microprogramma che guidi in modo efficiente le singole porte logiche non possiamo utilizzare l'interprete JVM di Sun; questo interprete è stato infatti scritto

in C per essere portabile e non può controllare l'hardware al livello di dettaglio che necessitiamo. Dato che l'hardware realmente utilizzato consiste solamente nei componenti elementari descritti nel Capitolo 3, in teoria il lettore, dopo aver compreso interamente il capitolo, potrebbe uscire di casa, farsi una bella scorta di transistor ed essere in grado di costruire da solo questo sottoinsieme della macchina JVM. Agli studenti che riusciranno a portare a termine il compito verrà assegnato un credito in più (e verrà offerta loro anche una seduta psichiatrica gratuita!).

Un modello convenzionale per progettare una microarchitettura consiste nel concepirla come un problema di programmazione, in cui ogni istruzione del livello ISA è una funzione che deve essere richiamata dal programma principale. In questo modello il programma principale è un semplice ciclo senza fine che determina la funzione da invocare, la richiama e poi ricomincia la propria esecuzione, come suggerito dalla Figura 2.3.

Il microprogramma ha delle variabili che costituiscono lo **stato** del calcolatore. Ogni funzione cambia il valore di almeno una delle variabili, modificando di conseguenza lo stato del calcolatore. Il *Program Counter* (PC, "contatore d'istruzioni") è una delle variabili che fanno parte dello stato e indica la locazione di memoria contenente la successiva funzione (cioè la successiva istruzione ISA) da eseguire. Durante l'esecuzione di un'istruzione il PC viene fatto avanzare in modo da farlo puntare all'istruzione successiva.

Le istruzioni IJVM sono corte e dall'aspetto semplice. Ogni istruzione è composta da alcuni campi, di solito uno o due, con uno scopo predefinito. Il primo campo di ogni istruzione è il **codice operativo** (*opcode*), che identifica il tipo d'istruzione, indicando se è di tipo `ADD`, di tipo `BRANCH` o altro. In molte istruzioni è presente un campo aggiuntivo che specifica l'operando: per esempio le istruzioni che accedono a una variabile locale devono indicare a *quale* variabile si riferiscono.

Questo modello di esecuzione, chiamato talvolta **fetch-esecuzione**, è utile a livello astratto e può anche costituire la base per l'implementazione di ISA complessi come IJVM. In seguito descriveremo come funziona questo modello, che aspetto ha la sua microarchitettura e com'è controllato dalle microistruzioni. L'insieme delle microistruzioni compone il microprogramma e ciascuna di loro ha il controllo del percorso dati durante un ciclo. Nel corso del capitolo presenteremo e tratteremo in modo dettagliato il microprogramma.

4.1.1 Percorso dati

Il **percorso dati** è quella parte della CPU che contiene la ALU, i suoi input e i suoi output. Il percorso dati del nostro esempio è mostrato nella Figura 4.1. Anche se è stato attentamente ottimizzato per interpretare i programmi IJVM esso è tuttavia molto simile ai percorsi dati della maggior parte delle macchine. Il nostro percorso dati contiene dei registri a 32 bit, a cui abbiamo assegnato nomi come `PC`, `SP` e `MDR`. Alcuni di questi nomi sono già familiari, ma è importante capire che è possibile accedere a questi registri solamente a livello di microarchitettura (cioè dal microprogramma). Il motivo per cui sono stati assegnati tali nomi è che generalmente questi registri memorizzano il valore di una variabile omonima appartenente all'architettura del livello ISA. La maggior parte dei registri può inviare il proprio contenuto sul bus B, collegato in input alla ALU. L'output della ALU guida invece lo *shifter*, che a sua volta invia il proprio risultato sul bus C; i valori di quest'ultimo possono essere scritti allo stesso tempo in uno o più registri. Per il momento non è presente il bus A (che verrà aggiunto in una fase successiva).

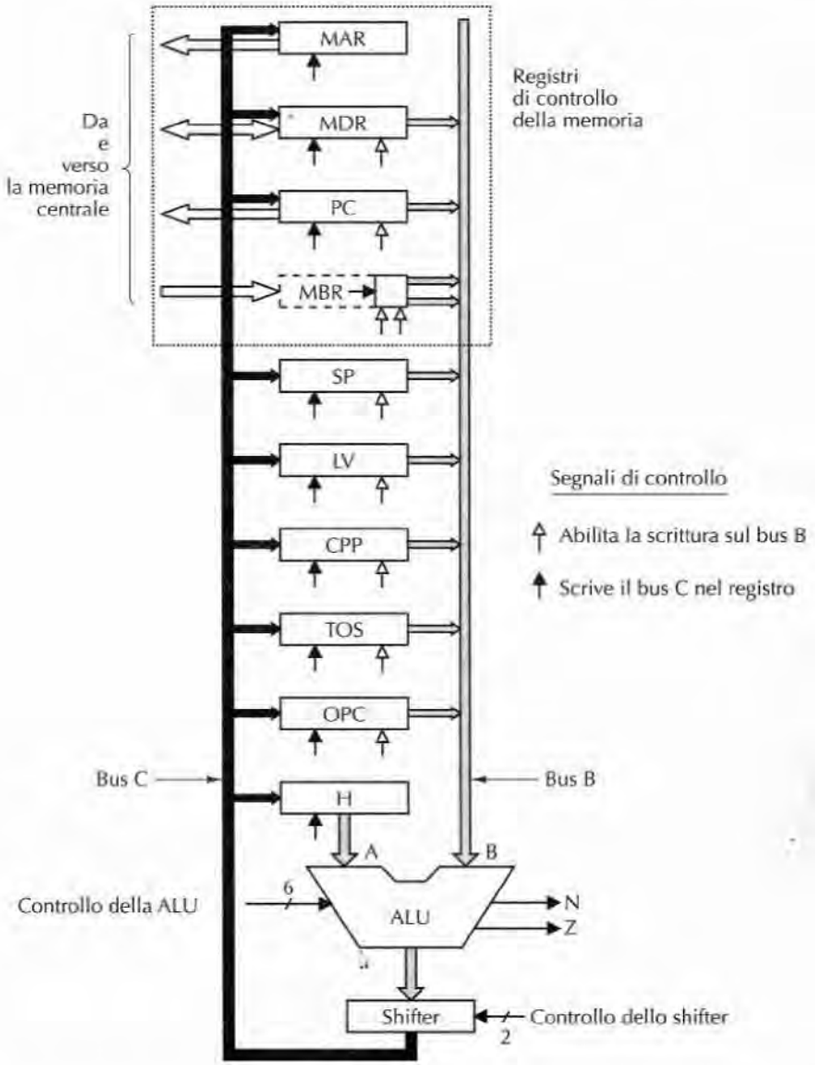


Figura 4.1 Percorso dati della microarchitettura utilizzata per l'esempio di questo capitolo.

La ALU è identica a quella vista nelle Figure 3.19 e 3.20, e la sua funzione è determinata da sei linee di controllo. Nella Figura 4.1 il trattino diagonale con a fianco il numero "6" indica che ci sono sei linee per il controllo della ALU. Fra queste F_0 e F_1 determinano l'operazione della ALU, `ENA` e `ENB` abilitano individualmente i due input, `INVA` inverte l'input di sinistra e `INC` forza la presenza di un riporto nel bit meno significativo, sommando quindi 1 al risultato. Non tutte le 64 combinazioni delle linee della ALU hanno un ruolo significativo.

F_0	F_1	ENA	ENB	INVA	INC	Funzione
0	1	1	0	0	0	A
0	1	0	1	0	0	B
0	1	1	0	1	0	$\overline{A}$
1	0	1	1	0	0	$\overline{B}$
1	1	1	1	0	0	A + B
1	1	1	1	0	1	A + B + 1
1	1	1	0	0	1	A + 1
1	1	0	1	0	1	B + 1
1	1	1	1	1	1	B − A
1	1	0	1	1	0	B − 1
1	1	1	0	1	1	−A
0	0	1	1	0	0	A AND B
0	1	1	1	0	0	A OR B
0	1	0	0	0	0	0
1	1	0	0	0	1	1
1	1	0	0	1	0	−1

Figura 4.2 Segnali di controllo in ingresso alla ALU e corrispondenti funzioni.

La Figura 4.2 mostra alcune delle combinazioni più interessanti. Non tutte queste funzioni sono necessarie per IJVM, ma molte torneranno utili quando tratteremo l'intero insieme JVM. In molti casi esistono inoltre molteplici possibilità per raggiungere lo stesso risultato. Nello schema + e − indicano la somma e la sottrazione aritmetica; $-A$ indica il complemento a due di A.

La ALU della Figura 4.1 richiede due dati in ingresso: un input sinistro (A) e uno destro (B). A quello di sinistra è collegato un registro H di mantenimento (*holding*), mentre al registro di destra è collegato il bus B. Quest'ultimo può essere caricato con i valori di una qualsiasi delle nove sorgenti indicate dalle nove frecce grigie la cui punta tocca il bus. Nel corso del capitolo tratteremo un'architettura di tipo differente, basata su due bus, che ci costringerà a effettuare nuove scelte e nuovi bilanciamenti.

È possibile caricare un valore in H scegliendo una funzione della ALU il cui compito sia semplicemente quello di far passare al suo interno l'input di destra (proveniente dal bus B) per poi porlo in output senza alcuna modifica. Una simile funzione può essere ottenuta attraverso la somma di due input della ALU negando però il segnale ENA, di modo che l'input di sinistra sia forzato al valore zero. Sommando zero all'input proveniente dal bus B si ottiene come risultato lo stesso valore presente sul bus B. Per poter memorizzare questo risultato in H lo si può far passare attraverso lo shifter senza fargli subire alcuna modifica.

Oltre alle funzioni appena citate, per gestire l'output della ALU è possibile utilizzare altre due linee di controllo. SLL8 (*Shift Left Logical*, "scorrimento logico a sinistra") trasla il valore a sinistra di un byte, impostando gli 8 bit meno significativi a 0. SRA1 (*Shift Right Arithmetic*, "scorrimento aritmetico a destra") trasla invece il valore di 1 bit a destra, lasciando inalterato il bit più significativo.

È possibile leggere o scrivere esplicitamente lo stesso registro durante un unico ciclo. Per incrementare SP di 1 è possibile portare il valore di SP sul bus B, disabilitare l'input sinistro della ALU, abilitare il segnale INC e memorizzare il risultato nuovamente all'interno di SP. Com'è possibile leggere e scrivere un registro nello stesso ciclo senza generare dei dati incoerenti? La soluzione risiede nel fatto che la lettura e la scrittura sono in realtà eseguite in momenti diversi all'interno del ciclo. Quando si seleziona un registro come input destro della ALU i suoi valori vengono inseriti nel bus B in una fase iniziale del ciclo e vengono poi continuamente mantenuti sul bus per tutta la durata del ciclo. La ALU compie quindi le proprie operazioni generando un risultato che giunge al bus C passando attraverso lo shifter. Verso la fine del ciclo, quando gli output della ALU e dello shifter sono sicuramente stabili, un segnale di clock dà inizio alla memorizzazione del contenuto del bus C all'interno di uno o più registri. Uno di questi registri potrebbe tranquillamente essere lo stesso dal quale proveniva l'input del bus B. Seguendo la modalità appena descritta, che sfrutta la precisa temporizzazione del percorso dati, è possibile leggere e scrivere lo stesso registro durante un solo ciclo.

Temporizzazione del percorso dati

La Figura 4.3 mostra la temporizzazione degli eventi sul percorso dati. All'inizio di ogni ciclo di clock viene generato un breve impulso. Come mostrato nella Figura 3.21(c), questo impulso può essere determinato dal clock principale. In corrispondenza del fronte di discesa dell'impulso vengono impostati i bit che piloteranno tutte le porte logiche. Quest'operazione richiede un intervallo di tempo finito e conosciuto a priori: Δw. Il registro richiesto viene quindi selezionato e il suo contenuto viene portato sul bus B; prima che il suo valore diventi

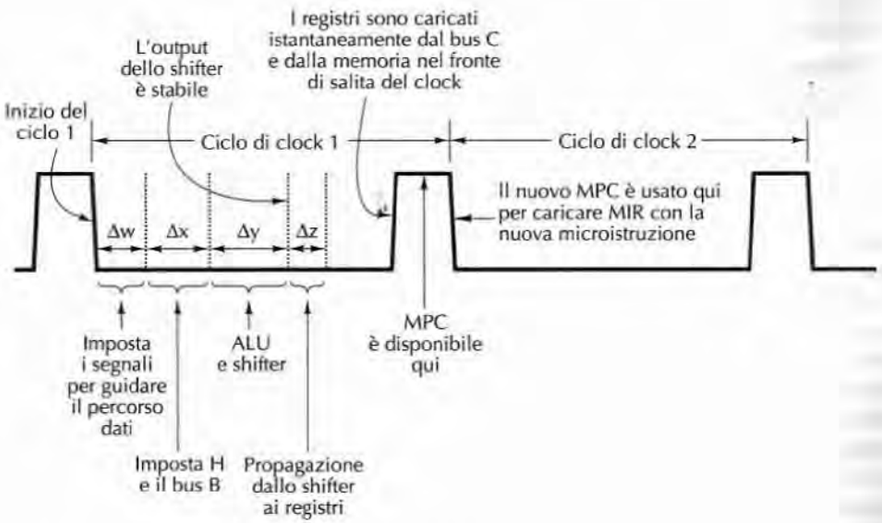


Figura 4.3 Temporizzazione di un ciclo di clock del percorso dati.

stabile occorre attendere un tempo Δx. A questo punto la ALU e lo shifter cominciano a operare sui dati validi e i loro output diventano stabili dopo un altro intervallo temporale, Δy. Passato un ulteriore tempo Δz i risultati vengono propagati lungo il bus C fino ai registri in cui possono essere caricati in corrispondenza del fronte di salita dell'impulso successivo. Il caricamento all'interno dei registri dovrebbe essere pilotato dal fronte del segnale ed essere veloce; in questo modo, anche se alcuni dei registri di input vengono modificati, gli effetti di queste modifiche giungeranno sul bus C solo dopo un tempo sufficientemente lungo rispetto al momento in cui sono stati caricati i registri. Inoltre, in corrispondenza del fronte di salita dell'impulso, il registro che stava alimentando il bus B smette di farlo, in preparazione del ciclo successivo. Nella figura sono indicati MPC, MIR e la memoria; i loro ruoli saranno presentati a breve.

È importante rendersi conto che all'interno del percorso dati esiste un tempo di propagazione finito, anche se non sono presenti elementi di memorizzazione. Modificare il valore sul bus B non modifica il bus C se non dopo un intervallo di tempo finito (dovuto ai ritardi che vengono introdotti a ogni passo). Di conseguenza, anche se un'operazione di scrittura modifica uno dei registri di input, il valore sarà reinserito in modo sicuro al suo interno, molto tempo prima che il valore (non più corretto) che si sta inserendo sul bus B (oppure H) possa raggiungere la ALU.

Per far sì che questa architettura funzioni è necessaria una rigida temporizzazione, un lungo ciclo di clock, un ritardo di propagazione attraverso la ALU conosciuto a priori e un veloce caricamento dei registri dal bus C. Pur essendo un compito complesso, grazie a un'attenta opera di ingegneria è possibile progettare il percorso dati in modo che funzioni correttamente in ogni momento. Le macchine reali funzionano proprio in questa maniera.

Un modo diverso per vedere il ciclo del percorso dati consiste nel pensare che esso sia implicitamente diviso in più sottocicli e che l'inizio del sottociclo 1 sia guidato dal fronte di discesa del clock. Di seguito sono elencate le attività che si svolgono durante i sottocicli, accompagnate (tra parentesi) dalla durata del sottociclo corrispondente.

1. Si impostano i segnali di controllo (Δw).
2. I registri vengono caricati nel bus B (Δx).
3. La ALU e lo shifter svolgono le loro operazioni (Δy).
4. I risultati vengono propagati lungo il bus C e ritornano nei registri (Δz).

In corrispondenza del fronte di salita del ciclo successivo i risultati vengono memorizzati nei registri.

Abbiamo detto che è possibile pensare ai sottocicli come se fossero definiti *implicitamente*. Con questo vogliamo intendere che nessun impulso di clock o altro segnale esplicito comunica alla ALU quando funzionare né dice ai risultati di entrare nel bus C. In realtà la ALU e lo shifter funzionano in continuazione; tuttavia i loro input vanno considerati come inconsistenti fino al tempo $\Delta w + \Delta x$ dopo il fronte di discesa del clock. Analogamente anche i loro output sono inconsistenti finché non sia trascorso un tempo $\Delta w + \Delta x + \Delta y$ dopo il fronte di discesa del clock. Gli unici segnali espliciti che guidano il percorso dati sono il fronte di discesa del clock, che fa partire il ciclo del percorso dati, e quello di salita, che carica i registri dal bus C. I limiti degli altri sottocicli sono determinati implicitamente dai tempi di propagazione insiti nei circuiti utilizzati. È responsabilità del progettista assicurarsi che il tempo $\Delta w + \Delta x + \Delta y + \Delta z$ giunga sufficientemente in anticipo rispetto al fronte di salita del clock, in modo che il caricamento dei registri possa funzionare in ogni momento.

Operazioni della memoria

La nostra macchina ha due modi diversi per comunicare con la memoria: una porta a 32 bit con indirizzi espressi in parole e una porta a 8 bit con indirizzi espressi in byte. Come mostra la Figura 4.1, la porta a 32 bit è controllata da due registri, MAR (*Memory Address Register*, "registro degli indirizzi di memoria") e MDR (*Memory Data Register*, "registro dei dati di memoria"). La porta a 8 bit è controllata invece da un unico registro, PC, che legge 1 byte negli 8 bit meno significativi di MBR. Questa porta può soltanto leggere i dati dalla memoria.

Ciascuno dei registri (così come tutti gli altri della Figura 4.1) è comandato da uno o due **segnali di controllo**. Una freccia bianca sotto un registro indica un segnale di controllo che abilita l'output del registro verso il bus B. Dato che MAR non ha una connessione al bus B, esso non ha il segnale per l'abilitazione. Neanche H ne è provvisto, dato che è l'unico possibile input sinistro della ALU ed è quindi sempre abilitato.

Una freccia nera sotto un registro indica un segnale di controllo che scrive nel registro (cioè "carica") un valore proveniente dal bus C. Dato che MBR non può essere caricato dal bus C, non ha un segnale di scrittura (anche se dispone di altri due segnali, descritti più avanti). Per iniziare una lettura o una scrittura occorre caricare il registro di memoria appropriato e successivamente inviare alla memoria un segnale di scrittura (non mostrato nella Figura 4.1).

MAR contiene gli indirizzi espressi in parole, in cui i valori 0, 1, 2, e così via, si riferiscono quindi a parole consecutive. PC contiene invece gli indirizzi espressi in byte e quindi i valori 0, 1, 2, e così via, fanno riferimento a byte consecutivi. Se si inserisce in PC il valore 2 e si fa partire un'operazione di lettura, il byte 2 della memoria verrà letto e inserito negli 8 bit meno significativi di MBR. Se invece si inserisce il valore 2 in MAR e si fa partire una lettura, saranno i byte 8–11 (cioè la parola 2) della memoria a essere letti e inseriti in MDR.

Queste due diverse modalità di accesso sono necessarie poiché MAR e PC saranno utilizzati per far riferimento a due parti diverse della memoria. In seguito risulterà più chiaro il motivo di questa distinzione, ma per il momento basta dire che la combinazione MAR/MDR è utilizzata per leggere e scrivere parole di dati del livello ISA, mentre la combinazione PC/MBR è utilizzata per leggere il programma eseguibile del livello ISA, che consiste in un flusso di byte. Tutti gli altri registri contenenti indirizzi, come MAR, utilizzano indirizzi espressi in parole.

Nelle reali implementazioni è presente un'unica memoria, orientata al byte. Attraverso un semplice espediente è possibile consentire a MAR di contare il numero di parole (cosa necessaria per il modo in cui JVM è definita) anche se gli indirizzi della memoria fisica sono espressi in byte. Quando MAR viene portato sul bus degli indirizzi i suoi 32 bit non vengono mappati direttamente sulle 32 linee, da 0 a 31. Al contrario il bit 0 di MAR viene collegato alla linea 2 del bus degli indirizzi, il bit 1 di MAR viene collegato alla linea 3 del bus degli indirizzi e così via. I 2 bit più alti di MAR vengono scartati, dato che sono necessari soltanto per indirizzi superiori a 2^{32}, nessuno dei quali è significativo per la nostra macchina che ha

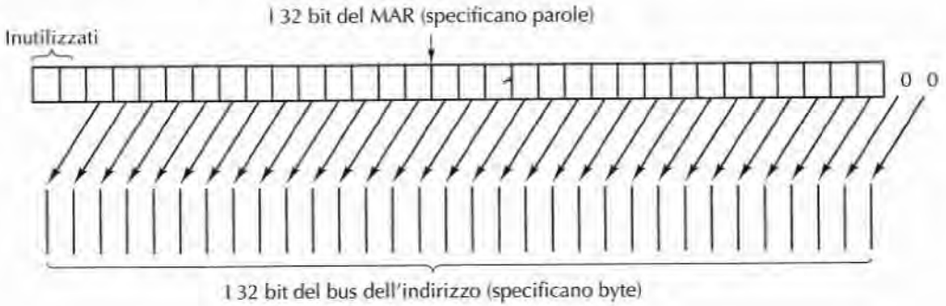


Figura 4.4 Corrispondenza tra i bit di MAR e i bit del bus dell'indirizzo.

un limite d'indirizzamento di 4 GB. In tal modo, quando MAR vale 1, viene posto sul bus l'indirizzo 4; quando MAR vale 2, viene posto l'indirizzo 8, e così via. Questo espediente è illustrato nella Figura 4.4.

Com'è già stato menzionato i dati letti dalla memoria attraverso la porta a 8 bit sono restituiti all'interno di MBR, un registro a 8 bit. MBR può essere copiato nel bus B in due modi distinti: con o senza segno. Quando si richiede un valore senza segno la parola a 32 bit inserita nel bus B contiene il valore di MBR negli 8 bit meno significativi, mentre i restanti 24 bit sono impostati a 0. I valori senza segno sono utili come indici di tabelle oppure quando occorre assemblare un intero a 16 bit a partire da 2 byte consecutivi (e senza segno) del flusso dati dell'istruzione.

L'altra possibilità per convertire il registro MBR a 8 bit in una parola a 32 bit consiste nel trattarlo come un valore con segno compreso tra –128 e +127 e utilizzare questo numero per generare una parola a 32 bit che abbia lo stesso valore numerico. Questa conversione viene effettuata mediante un procedimento chiamato **estensione del segno** che consiste nel duplicare il bit del segno (quello più a sinistra) di MBR nei 24 bit più alti del bus B. Quando si sceglie questa tecnica i 24 bit più alti saranno tutti 0 oppure tutti 1, a seconda che il bit più a sinistra di MBR valga 0 oppure 1.

La scelta tra convertire gli 8 bit di MBR in un valore a 32 bit con o senza segno prima di copiarlo sul bus B è determinata dal segnale di controllo (indicato nella Figura 4.1 dalle frecce bianche sotto MBR) che è asserito. La necessità di distinguere tra queste due opzioni giustifica la presenza delle due frecce. Il rettangolo tratteggiato che nella figura si trova alla sinistra di MBR indica che è possibile far sì che il registro MBR a 8 bit si comporti come una sorgente a 32 bit del bus B.

4.1.2 Microistruzioni

Per controllare il percorso dati della Figura 4.1 abbiamo bisogno di 29 segnali suddivisibili in cinque gruppi funzionali:

- 9 segnali per controllare la scrittura dei dati dal bus C all'interno dei registri
- 9 segnali per controllare l'abilitazione dei registri sul bus B per l'input della ALU
- 8 segnali per controllare le funzioni della ALU e dello shifter

- 2 segnali (non mostrati) per indicare alla memoria di leggere (scrivere) attraverso MAR (MDR)
- 1 segnale (non mostrato) per indicare il prelievo dalla memoria attraverso PC o MBR.

I valori di questi 29 segnali specificano le operazioni da eseguire durante un ciclo del percorso dati. Un ciclo consiste nel portare i valori dei registri sul bus B, propagare i segnali attraverso la ALU e lo shifter, guidarli sul bus C e infine scrivere i risultati nel registro o nei registri appropriati. Inoltre, nel caso in cui sia asserito il segnale per una lettura dalla memoria, l'operazione viene fatta iniziare alla fine del ciclo del percorso dati, dopo che MAR è stato caricato. Alla fine del ciclo *seguente* i dati della memoria sono disponibili in MBR oppure in MDR e sono utilizzabili nel ciclo *ancora successivo*. In altre parole una lettura della memoria (su una delle due porte) iniziata alla fine del ciclo k trasmette dati che non possono essere utilizzati nel ciclo $k + 1$, ma soltanto a partire dal ciclo $k + 2$.

Questo comportamento apparentemente contraddittorio è spiegato nella Figura 4.3. Durante il ciclo 1 i segnali di controllo della memoria vengono generati soltanto verso la fine del ciclo, subito dopo il momento in cui MAR e PC sono caricati in corrispondenza del fronte di salita del clock. Assumeremo che la memoria inserisca i propri risultati nei bus di memoria entro un ciclo, di modo che MBR e/o MDR possano essere caricati, insieme a tutti gli altri registri, nel successivo fronte di salita del clock.

Detto in altre parole, carichiamo MAR alla fine del ciclo del percorso dati e poco dopo facciamo partire l'operazione di memoria. Di conseguenza non possiamo aspettarci che i risultati di un'operazione di lettura siano già disponibili in MDR all'inizio del ciclo successivo, soprattutto se la larghezza dell'impulso di clock è breve. Se la memoria richiede un ciclo di clock non c'è tempo a sufficienza; deve per forza passare un ciclo del percorso dati tra l'inizio di una lettura della memoria e l'utilizzo del risultato. Ovviamente durante questo ciclo è possibile eseguire altre operazioni, a patto che queste non necessitino di parole dalla memoria.

L'ipotesi che la memoria richieda un ciclo per eseguire la propria operazione è equivalente ad assumere che la frequenza di successi della cache di primo livello sia pari al 100%. Questa assunzione non è mai vera, ma, per gli scopi che ci siamo posti, sarebbe troppo complesso considerare un tempo di ciclo della memoria di durata variabile.

Dato che MBR e MDR sono caricati insieme a tutti gli altri registri in corrispondenza del fronte di salita del clock, essi possono essere letti nei cicli in cui si sta svolgendo una nuova lettura della memoria. Essi restituiscono valori vecchi in quanto la lettura della memoria non ha ancora avuto il tempo di sovrascriverli e aggiornarli. Tuttavia questa situazione non presenta ambiguità; finché i nuovi valori non siano caricati in MBR e MDR nel fronte di salita del clock, i valori precedenti sono ancora presenti e utilizzabili. Dato che una lettura richiede solamente un ciclo, è possibile eseguire letture in sequenza durante due cicli consecutivi. È possibile inoltre utilizzare nello stesso momento entrambe le porte di memoria, anche se tentare di leggere e scrivere simultaneamente lo stesso byte genera risultati indefiniti.

In alcuni casi può essere utile scrivere l'output presente nel bus C in più di un registro, mentre in nessun caso ha senso abilitare nello stesso momento più di un registro sul bus B (in alcuni casi reali ciò potrebbe addirittura essere dannoso). Con pochi circuti aggiuntivi è possibile ridurre il numero di bit necessari per la selezione delle sorgenti che alimentino il

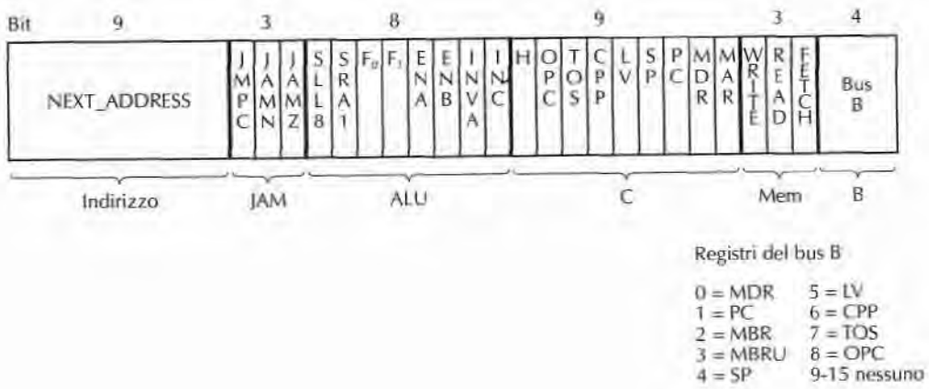


Figura 4.5 Formato delle microistruzioni di Mic-1.

bus B. Ci sono soltanto nove possibili registri di input che possono guidare il bus B (considerando separatamente le versioni con e senza segno di MBR). Possiamo quindi codificare in 4 bit l'informazione del bus B e utilizzare un decodificatore per generare 16 segnali di controllo, 7 dei quali non vengono utilizzati. Se si trattasse di un progetto commerciale gli ingegneri subirebbero forti pressioni dai loro capi per costringerli a sbarazzarsi di uno dei registri, in modo da rendere sufficienti soltanto 3 bit. In quanto accademici noi possiamo permetterci l'enorme lusso di sprecare 1 bit semplicemente per ottenere un'architettura più semplice e ordinata.

A questo punto possiamo controllare il percorso dati con 9 + 4 + 8 + 2 + 1 = 24 segnali, quindi con 24 bit. Tuttavia questi bit controllano il percorso dati soltanto per un ciclo. La seconda parte del controllo consiste invece nel determinare che cosa deve essere effettuato durante il ciclo successivo. Per includere questo aspetto nel progetto del controllore definiremo un formato che ci permetterà di descrivere le operazioni da eseguire utilizzando i 24 bit di controllo più due campi aggiuntivi: NEXT_ADDRESS e JAM. La Figura 4.5 mostra un possibile formato, diviso in sei gruppi e contenente i 36 segnali seguenti.

`Addr` – Contiene l'indirizzo di una potenziale successiva microistruzione.

`JAM` – Determina come viene selezionata la successiva microistruzione.

`ALU` – Seleziona le funzioni della ALU e dello shifter.

`C` – Seleziona quali registri sono scritti dal bus C.

`Mem` – Seleziona la funzione della memoria.

`B` – Seleziona la sorgente del bus B; la codifica è mostrata nella figura.

In teoria l'ordinamento dei gruppi è arbitrario, anche se in realtà quello della Figura 4.6 è stato scelto attentamente in modo da minimizzare l'incrocio fra le linee. Nei diagrammi schematici simili alla Figura 4.6 l'incrocio fra linee spesso corrisponde a collegamenti che si incrociano sui chip. Dato che causano difficoltà nei progetti è buona norma cercare di minimizzarli.

4.1.3 Unità di controllo microprogrammata: Mic-1

Finora abbiamo descritto come viene controllato il microprogramma, ma non abbiamo ancora spiegato come si decide quale dei segnali di controllo abilitare durante ciascun ciclo. Ciò è determinato da un **sequenzializzatore** che ha la responsabilità di far avanzare passo passo la sequenza di operazioni necessarie per eseguire una singola istruzione ISA.

Durante ogni ciclo il sequenzializzatore deve produrre due tipi d'informazione:

1. lo stato di ogni segnale di controllo del sistema
2. l'indirizzo della microistruzione da eseguire subito dopo.

La Figura 4.6 è un diagramma a blocchi dettagliato dell'intera microarchitettura della nostra macchina di esempio, che chiameremo **Mic-1**. A prima vista potrebbe sembrare imponente, ma vale la pena studiarlo attentamente. Quando avremo compreso il significato di tutti i rettangoli e di tutte le linee della figura, saremo sulla giusta strada per una piena comprensione del livello di microarchitettura. Il diagramma a blocchi è composto da due parti principali: il percorso dati, sulla sinistra, già approfonditamente analizzato e la sezione di controllo, sulla destra, che considereremo a partire da questo momento.

L'elemento più grande e più importante della sezione di controllo è una memoria chiamata **memoria di controllo**. Anche se a volte viene implementata come un insieme di porte logiche, può essere utile pensarla come una memoria che contiene l'intero microprogramma. In generale ci riferiremo a essa con il nome di memoria di controllo per non confonderla con la memoria principale, a cui si accede mediante i registri MBR e MDR. In ogni caso, dal punto di vista funzionale, la memoria di controllo è semplicemente un circuito che memorizza le microistruzioni invece che le istruzioni ISA. Nel nostro caso contiene 512 parole, consistenti in una microistruzione a 36 bit del tipo mostrato nella Figura 4.5. In realtà non sono necessarie tutte queste parole, ma (per ragioni che spiegheremo tra poco) abbiamo bisogno di indirizzi per 512 parole distinte.

La memoria di controllo differisce dalla memoria centrale per un aspetto importante: le istruzioni della memoria centrale sono sempre eseguite nell'ordine determinato dagli indirizzi (tranne nel caso delle diramazioni), mentre così non è nel caso delle microistruzioni. Nella Figura 2.3 incrementare il contatore di programma significa che la successiva istruzione da eseguire corrisponde a quella che segue l'istruzione corrente all'interno della memoria.

I microprogrammi richiedono maggior flessibilità (dato che le sequenze delle microistruzioni tendono a essere brevi) e quindi, solitamente, ogni microistruzione specifica in modo esplicito il proprio successore.

Anche la memoria di controllo necessita di un proprio registro di indirizzo e di un proprio registro dei dati (ma non dei segnali di lettura e scrittura dato che la memoria viene letta in continuazione). Chiameremo **MPC** (*MicroProgram Counter*) il registro degli indirizzi della memoria di controllo. Questo nome è incongruo, dato che le locazioni non seguono un ordine preciso; esse sono specificate in modo esplicito e quindi non c'è bisogno di effettuare alcun conteggio (ma come si fa a mettere in discussione la tradizione?). Il registro dei dati della memoria viene invece chiamato **MIR** (*MicroInstruction Register*, "registro della microistruzione corrente"). Il suo ruolo consiste nel memorizzare la microistruzione in corso di esecuzione, i cui bit determinano i segnali di controllo che guidano il percorso dati.

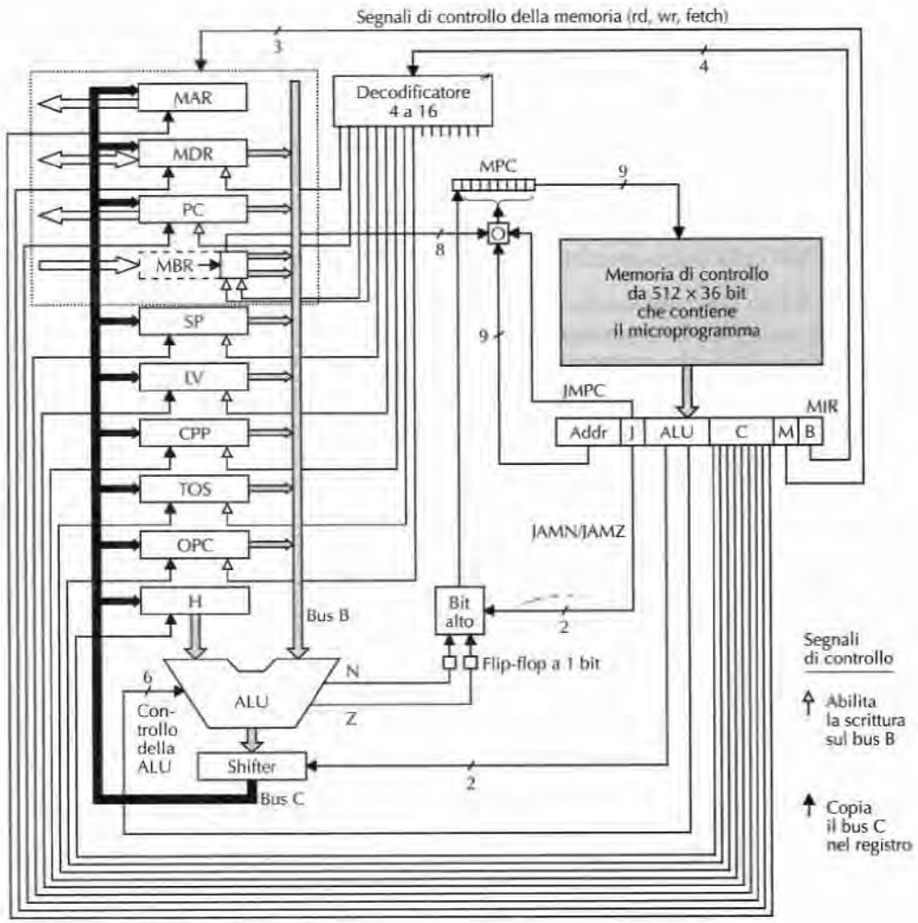


Figura 4.6 Diagramma a blocchi del nostro esempio di microarchitettura di Mic-1.

Nella Figura 4.6 il registro `MIR` contiene gli stessi sei gruppi mostrati nella Figura 4.5. I gruppi `Addr` e `J` (iniziale di `JAM`) controllano la selezione della microistruzione successiva e verranno illustrati a breve. Il gruppo della ALU contiene 8 bit che selezionano la funzione della ALU e guidano lo shifter. I bit `C` indicano in quali registri verrà caricato l'output della ALU dal bus C. I bit `M` controllano le operazioni della memoria.

Infine gli ultimi 4 bit guidano il decodificatore, che determina quale registro deve essere portato sul bus B. Nel nostro esempio abbiamo scelto di utilizzare un decodificatore standard da 4 a 16, anche se servono solamente nove possibilità. In un progetto ottimizzato anche nei minori dettagli si sarebbe utilizzato un decodificatore da 4 a 9. In questo caso si è deciso di utilizzare un circuito standard già disponibile invece di progettarne

appositamente uno nuovo. L'utilizzo di circuiti standard è più semplice e meno soggetto a errori.

Il funzionamento (Figura 4.6) è il seguente. All'inizio di un ciclo di clock (il fronte di discesa del clock nella Figura 4.3) la parola contenuta nella memoria di controllo e puntata da `MPC` viene trasferita in `MIR`. Il tempo necessario per effettuare questa operazione è indicato nella figura con Δw. In termini di sottocicli, si può pensare che `MIR` venga caricato durante il primo sottociclo.

Subito dopo, i vari segnali si propagano all'interno del percorso dati. Grazie a questi segnali il contenuto di un registro viene inserito nel bus B e la ALU sa quale operazione deve eseguire. Seguono poi varie azioni, fino ad arrivare al secondo sottociclo. Dopo un intervallo di $\Delta w + \Delta x$ dall'inizio del ciclo, gli input della ALU diventano stabili.

Dopo un altro Δy, tutto nel circuito si è stabilizzato, compresi gli output della ALU, di `N`, di `Z` e dello shifter. I valori di `N` e `Z` vengono quindi salvati in una coppia di flip-flop. Questi bit, come tutti i registri caricati dal bus C, vengono salvati in corrispondenza del fronte di salita del clock, verso la fine del ciclo del percorso dati. L'output della ALU non viene memorizzato, ma semplicemente inserito nello shifter. Le attività della ALU e dello shifter si svolgono nel corso del sottociclo 3.

Dopo un ulteriore intervallo di tempo Δz, l'output dello shifter raggiunge i registri attraverso il bus C. I registri possono successivamente essere caricati verso la fine del ciclo (nel fronte di salita dell'impulso del clock della Figura 4.3). Nel sottociclo 4 vengono caricati i registri e i flip-flop `N` e `Z`. Il sottociclo termina leggermente dopo il fronte di salita del clock, quando tutti i risultati sono stati salvati, i risultati delle precedenti operazioni di memoria sono disponibili e `MPC` è stato caricato. Questo processo continua ripetutamente finché qualcuno non si stufi e spenga la macchina.

Il microprogramma, oltre a guidare il percorso dati, deve parallelamente determinare quale sarà la microistruzione successiva, dato che non è necessario che esse vengano eseguite nello stesso ordine in cui appaiono nella memoria di controllo. Il calcolo dell'indirizzo della microistruzione successiva comincia dopo che `MIR` è stato caricato ed è diventato stabile. Inizialmente i 9 bit del campo `NEXT_ADDRESS` vengono copiati all'interno di `MPC`. Mentre si svolge questa copia viene ispezionato il campo `JAM`; se il suo valore è 000, allora non viene eseguita alcuna azione aggiuntiva. Quando termina la copia di `NEXT_ADDRESS`, `MPC` punta alla microistruzione successiva.

Se invece uno o più bit di `JAM` valgono 1, allora è necessario compiere delle azioni. Se `JAMN` è abilitato, se ne calcola l'OR logico con il flip-flop `N` (1 bit) e si memorizza il risultato nel bit più significativo di `MPC`. Analogamente, se è abilitato `JAMZ`, si calcola l'OR tra questo segnale e il flip-flop `Z`. Se sono abilitati entrambi, si calcola invece l'OR rispetto a tutti e due. La ragion per cui sono necessari i flip-flop `N` e `Z` è che, dopo il fronte di salita del clock (mentre il clock è ancora alto), il bus B non viene più alimentato e quindi gli output della ALU non possono più essere considerati corretti. Salvando in `N` e `Z` i flag di stato della ALU è possibile rendere stabili questi valori, per poterli utilizzare per calcolare correttamente `MPC`, indipendentemente dalle operazioni che la ALU compie nel frattempo.

Nella Figura 4.6 i componenti logici che effettuano questa elaborazione sono etichettati con la scritta "Bit alto". La funzione booleana che calcola questo bit è la seguente:

```
F = (JAMZ AND Z) OR (JAMN AND N) OR NEXT_ADDRESS[8]
```

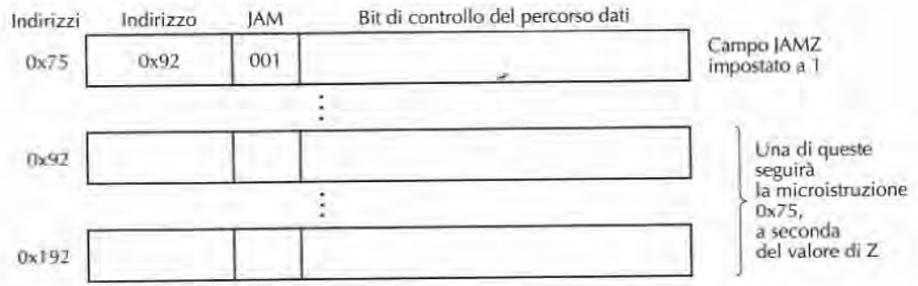


Figura 4.7 Le microistruzioni con JAMZ = 1 hanno due possibili successori.

Si noti che in tutti i casi possibili MPC può assumere soltanto uno di questi due valori:

1. il valore di NEXT_ADDRESS
2. il valore di NEXT_ADDRESS, in cui il bit più significativo è calcolato in OR con 1.

Non esistono altre possibilità. Se il bit più significativo di NEXT_ADDRESS valesse già 1, non avrebbe alcun senso utilizzare JAMN o JAMZ.

Si noti che quando tutti i bit di JAM valgono zero, l'indirizzo della successiva microistruzione da eseguire è semplicemente il numero a 9 bit contenuto nel campo NEXT_ADDRESS. In caso contrario, esistono due potenziali indirizzi per la microistruzione successiva: NEXT_ADDRESS e NEXT_ADDRESS calcolato in OR con 0x100 (assumendo che NEXT_ADDRESS ≤ 0xFF). (Si noti che 0x indica che il numero che lo segue è espresso in forma esadecimale.) Questo punto è illustrato nella Figura 4.7. La microistruzione corrente, che si trova nella locazione 0x75, ha il campo NEXT_ADDRESS = 0x92 e JAMZ impostato a 1. Di conseguenza l'indirizzo della microistruzione successiva dipende dal bit Z memorizzato durante la precedente operazione della ALU. Se il bit Z vale 0, la microistruzione successiva comincerà a partire dall'indirizzo 0x92; altrimenti, l'indirizzo sarà 0x192.

Il terzo bit del campo JAM è JMPC. Se è impostato a 1, gli 8 bit di MBR sono collegati in OR con gli 8 bit meno significativi del campo NEXT_ADDRESS della microistruzione precedente. Il risultato viene inviato a MPC. Il quadratino etichettato con "O" nella Figura 4.6 esegue l'OR di MBR e NEXT_ADDRESS quando JMPC vale 1, mentre lascia semplicemente passare NEXT_ADDRESS in MPC quando JMPC vale 0. In genere quando JMPC vale 1, gli 8 bit meno significativi di NEXT_ADDRESS valgono 0. Il bit più significativo può essere 0 oppure 1 e quindi il valore di NEXT_ADDRESS utilizzato con JMPC è generalmente 0x000 oppure 0x100. Il motivo per cui a volte si usa 0x000 mentre altre volte si usa 0x100 sarà spiegato in seguito.

La possibilità di calcolare l'OR di MBR e NEXT_ADDRESS e di memorizzare il risultato in MPC permette di implementare in modo efficiente una diramazione. Si noti che è possibile specificare uno qualsiasi dei 256 indirizzi unicamente sulla base dei bit presenti in MBR.

In un utilizzo tipico MBR contiene un codice operativo; in questo modo l'uso di JMPC fornisce, per ogni possibile codice operativo, un unico indirizzo in cui trovare la successiva microistruzione da eseguire. Questo metodo è utile per effettuare velocemente un salto verso la funzione associata al codice operativo appena prelevato.

Dato che la comprensione della temporizzazione della macchina è di fondamentale importanza per gli argomenti che seguiranno, vale la pena ripetere nuovamente il concetto. Questa volta lo faremo in termini di sottocicli, per permetterne una più facile visualizzazione; non dimentichiamoci però che gli unici reali eventi del clock sono il fronte di discesa, che fa partire il ciclo, e il fronte di salita, che carica i registri e i flip-flop N e Z. Si guardi per favore, ancora una volta, la Figura 4.3.

Durante il sottociclo 1, iniziato in corrispondenza del fronte di discesa del clock, l'indirizzo contenuto in quel momento in MPC viene caricato all'interno di MIR. Durante il sottociclo 2 i segnali si propagano da MIR verso l'esterno e il registro selezionato viene caricato sul bus B. Durante il sottociclo 3 la ALU e lo shifter svolgono le proprie operazioni e generano un risultato stabile. Durante il sottociclo 4 il bus C, i bus di memoria e i valori della ALU diventano stabili. In corrispondenza del fronte di salita si verificano vari eventi: il contenuto del bus C viene caricato nei registri, i flip-flop N e Z vengono caricati, e MBR e MDR ottengono i loro risultati dall'operazione di memoria iniziata alla fine del precedente ciclo del percorso dati (se presente). Non appena MBR diventa stabile, si carica MPC in previsione della successiva microistruzione. MPC ottiene quindi il proprio valore verso la metà dell'intervallo, quando il clock è alto, ma dopo che MBR/MDR sono diventati disponibili. Il momento esatto in cui caricare MPC potrebbe essere determinato dal livello del segnale (invece che dal fronte) oppure da un ritardo temporale fisso rispetto al fronte di salita del clock. Ciò che veramente conta è che MPC non venga caricato finché non siano pronti i registri da cui dipende (MBR, N e Z). Non appena il clock scende MPC può indirizzare la memoria di controllo dando così inizio a un nuovo ciclo.

Si noti che ogni ciclo è autocontenuto; esso specifica che cosa andrà sul bus B, quali operazioni la ALU e lo shifter dovranno effettuare, dove verrà memorizzato il bus C e, infine, quale dovrà essere il successivo valore di MPC.

Vale la pena fare un'ultima osservazione riguardo alla Figura 4.6. Finora abbiamo considerato MPC come un normale registro, con una capacità di 9 bit, che viene caricato mentre il clock è alto. In realtà non c'è bisogno di utilizzare un registro: tutti i suoi input possono alimentare direttamente la memoria di controllo. È sufficiente che siano inviati alla memoria di controllo in corrispondenza del fronte di discesa del clock, quando MIR viene selezionato e letto; non c'è alcun bisogno di memorizzare effettivamente questi segnali all'interno di MPC. Per questa ragione MPC può essere implementato come un **registro virtuale**, cioè il luogo in cui raccogliere dei segnali e che assomiglia più a un pannello elettronico che a un vero e proprio registro. Il fatto di rendere MPC un registro virtuale semplifica la temporizzazione: in questo caso gli eventi si verificano solamente in corrispondenza dei fronti di salita e discesa del clock e in nessun altro momento. Se qualcuno ritiene che sia più facile interpretare MPC come un vero e proprio registro, può continuare a farlo, in quanto ciò non è altro che un diverso, ma comunque valido, punto di vista.

4.2 Esempio di ISA: IJVM

Continuiamo il nostro esempio introducendo il livello ISA della macchina che sarà interpretato dal microprogramma eseguito nella microarchitettura della Figura 4.6 (IJVM). Per praticità a volte ci riferiremo all'architettura dell'insieme d'istruzioni (ISA) con il termine di **macroarchitettura**, in contrapposizione con la microarchitettura. Prima di descrivere IJVM faremo una breve digressione che ci permetterà di metterne in luce le motivazioni.

4.2.1 Stack

Praticamente tutti i linguaggi di programmazione supportano il concetto di procedure (metodi), dotate di un insieme di variabili locali. È possibile accedere a queste variabili dall'interno della procedura, ma ciò diventa impossibile una volta che la procedura termina. La domanda che sorge è: "In quale parte della memoria bisogna memorizzare queste variabili?".

La soluzione più semplice, che consiste nell'assegnare a ciascuna variabile un indirizzo assoluto della memoria, non funziona. Il problema nasce dal fatto che una procedura potrebbe richiamare se stessa; nel corso del Capitolo 5 studieremo le procedure che si comportano in questo modo e che sono dette ricorsive. Per il momento è sufficiente dire che se una procedura è attiva (cioè è stata invocata) due volte, è impossibile memorizzare le sue variabili in locazioni assolute della memoria in quanto la seconda invocazione interferirebbe con la prima.

Si usa quindi un'altra strategia. Per memorizzare le variabili viene riservata un'area della memoria, chiamata **stack** ("pila"), al cui interno però non si stabiliscono indirizzi assoluti per le singole variabili. Si imposta invece un registro, chiamiamolo LV, in modo che punti alla procedura corrente. Nella Figura 4.8(a) è stata richiamata una procedura *A*, che possiede tre variabili locali, *a1*, *a2* e *a3*; per memorizzarle è stato quindi riservato uno spazio di memoria a partire dalla locazione puntata da LV. Un altro registro, SP, punta alla parola che si trova nella locazione più alta all'interno dello stack delle variabili locali di *A*. Se LV è 100 e le parole sono di 4 byte, il valore di SP sarà 108. Per far riferimento alle variabili locali si fornisce il loro spiazzamento (*offset*) rispetto a LV. La struttura dati compresa tra LV e SP (considerando anche le parole puntate dai due registri) è chiamata **blocco delle variabili locali** di *A*.

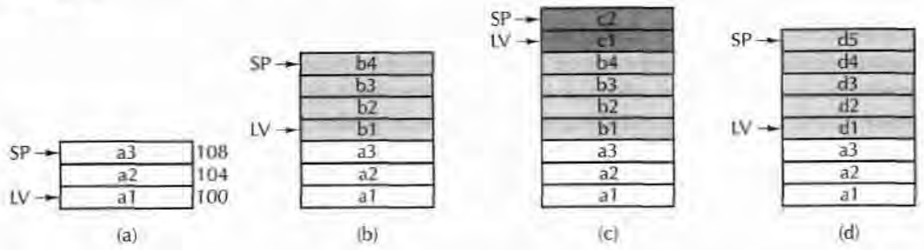


Figura 4.8 Utilizzo dello stack per memorizzare le variabili locali. (a) Mentre *A* è attiva. (b) Dopo che *A* ha richiamato *B*. (c) Dopo che *B* ha richiamato *C*. (d) Dopo che *C* e *B* hanno restituito il controllo e *A* ha richiamato *D*.

Consideriamo ora che cosa succede se *A* richiama un'altra procedura, diciamo *B*. Dove dovrebbero essere memorizzate le quattro variabili locali (*b1, b2, b3, b4*) di *B*? Risposta: nello stack sopra il blocco di *A*, come mostra la Figura 4.8(b). Si noti che LV è stato modificato dalla chiamata alla procedura in modo da puntare alle variabili locali di *B* invece che a quelle di *A*. È possibile far riferimento alle variabili locali di *B* fornendo il loro spiazzamento rispetto a LV. Analogamente, se *B* richiama *C*, LV e SP vengono nuovamente modificati in modo da allocare lo spazio necessario per memorizzare le due variabili di *C*; questa situazione è mostrata nella Figura 4.8(c).

Quando *C* termina, *B* diventa nuovamente attiva e lo stack viene modificato per tornare allo stato mostrato nella Figura 4.8(b), in modo che LV torni a puntare alle variabili locali di *B*. Analogamente, quando *B* restituisce il controllo, si torna nuovamente alla situazione mostrata nella Figura 4.8(a). In tutte le situazioni LV punta alla base del blocco dello stack corrispondente alla procedura correntemente attiva, mentre SP punta alla cima dello stesso blocco.

Supponendo ora che *A* richiami *D*, che ha cinque variabili locali, si ottiene la situazione mostrata nella Figura 4.8(d), che illustra come le variabili locali di *D* usino la stessa area di memoria utilizzata precedentemnte da *B*, oltre a una parte di quella che era stata utilizzata da *C*. Organizzando la memoria in questo modo, è possibile allocare solamente la memoria necessaria alle procedure attive. Quando una procedura restituisce il controllo viene rilasciata la memoria utilizzata dalle sue variabili locali.

Oltre a memorizzare le variabili locali gli stack hanno anche un altro utilizzo. Possono essere utilizzati per memorizzare gli operandi durante il calcolo di un'espressione aritmetica. Quando uno stack è utilizzato in questo modo ci si riferisce a esso con il termine di **stack degli operandi**. Supponiamo per esempio che prima di richiamare *B*, *A* debba eseguire il calcolo

```
a1 = a2 + a3;
```

Un modo per effettuare questa somma consiste nel porre *a2* in cima allo stack, come mostra la Figura 4.9(a). Qui SP è stato incrementato del numero di byte che formano una parola, diciamo 4, in modo da puntare all'indirizzo in cui è stato memorizzato il primo operando. In seguito *a3* viene posto in cima allo stack, come mostra la Figura 4.9(b). Aprendo una parentesi riguardo alla notazione utilizzata nel testo, specifichiamo che tutti i frammenti di programmi saranno scritti utilizzando il carattere `Courier`, com'è già stato fatto in precedenza. Utilizzeremo questo carattere anche per i codici operativi del linguaggio assemblativo e per i registri della macchina, mentre nel testo scriveremo in *corsivo* i nomi delle variabili e delle procedure del programma. La differenza risiede nel fatto che le variabili e i nomi delle procedure sono scelti dall'utente, mentre i codici operativi e i nomi dei registri sono predefiniti.

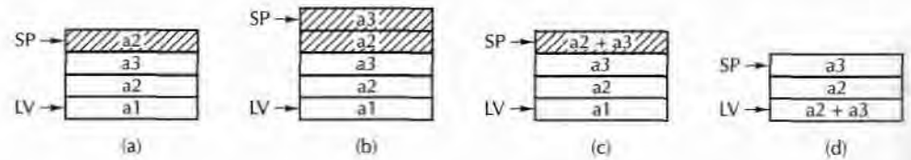


Figura 4.9 Utilizzo dello stack degli operandi per il calcolo di un'espressione aritmetica.

Il calcolo effettivo può a questo punto essere realizzato eseguendo un'istruzione che preleva due parole dallo stack, le somma e inserisce il risultato nuovamente nello stack, come mostra la Figura 4.9(c). Infine, la parola che si trova in cima allo stack può essere rimossa e memorizzata nella variabile locale *a1*, com'è illustrato nella Figura 4.9(d).

Il blocco delle variabili locali e lo stack degli operandi possono essere mischiati tra loro. Per esempio quando si calcola un'espressione come $x^2 + f(x)$, una parte dell'espressione (per esempio, x^2) può trovarsi nello stack degli operandi nel momento in cui viene invocata la funzione f. Il risultato della funzione viene lasciato nello stack, al di sopra di x^2, in modo che l'istruzione successiva li possa sommare.

Vale la pena precisare che mentre tutte le macchine utilizzano uno stack per memorizzare le variabili locali, non tutte usano uno stack degli operandi per eseguire le operazioni aritmetiche. In realtà la maggior parte non lo utilizza; JVM e IJVM funzionano però in questo modo e per questo motivo abbiamo deciso di introdurlo.

4.2.2 Modello della memoria di IJVM

Siamo ora pronti ad analizzare l'architettura di IJVM. Fondamentalmente essa consiste in una memoria concepibile in due modi distinti: come un array di 4.294.967.296 byte (4 GB) oppure come un array di 1.073.741.824 parole da 4 byte. Diversamente dalla maggior parte degli ISA, la Java Virtual Machine non rende direttamente visibili a livello ISA gli indirizzi di memoria assoluti, ma utilizza degli indirizzi impliciti che fornisono la base per l'uso di puntatori. L'unico modo che le istruzioni IJVM hanno per accedere alla memoria è quello di indicizzarla utilizzando questi puntatori. In ogni momento sono definite le seguenti aree di memoria.

1. *Porzione costante di memoria.* I programmi IJVM non possono scrivere in quest'area che contiene costanti, stringhe e puntatori ad altre aree di memoria cui è possibile far riferimento. È caricata quando il programma è portato in memoria e in seguito non viene modificata. Esiste un registro implicito, `CPP`, contenente gli indirizzi della prima parola della porzione costante di memoria.

2. *Blocco delle variabili locali.* Per ogni invocazione di un metodo viene allocata un'area in cui memorizzare le variabili locali durante l'intero ciclo di vita dell'invocazione. Quest'area è chiamata **blocco delle variabili locali**. Nella parte iniziale di questo blocco sono memorizzati i parametri (chiamati anche argomenti) con cui è stato invocato il metodo. Il blocco delle variabili locali non comprende lo stack degli operandi, che è separato. Tuttavia, per motivi di efficienza, qui abbiamo scelto di implementare lo stack degli operandi immediatamente sopra il blocco delle variabili locali. È presente un registro implicito, `LV`, contenente l'indirizzo della prima locazione all'interno del blocco delle variabili locali. I parametri passati durante l'invocazione del metodo sono memorizzati all'inizio del blocco delle variabili locali.

3. *Stack degli operandi.* Il blocco dello stack non può superare una certa dimensione, stabilita in anticipo dal compilatore Java. Lo spazio per lo stack degli operandi è allocato direttamente sopra il blocco delle variabili locali (Figura 4.10). Nella nostra implementazione conviene pensare che lo stack degli operandi faccia parte del blocco delle variabili locali. In ogni caso c'è un registro implicito contenente l'indirizzo della parola in cima allo stack. Si noti che, diversamente da `CPP` e `LV`, questo puntatore, `SP`, cambia durante l'esecuzione del metodo a seconda che gli operandi siano inseriti nello stack o rimossi da quest'ultimo.

4. *Area dei metodi.* Infine c'è una regione di memoria in cui risiede il programma, simile all'area "testo" di un processo UNIX. È presente un registro implicito che contiene l'indirizzo della successiva istruzione da prelevare. Questo puntatore è chiamato contatore di programma, oppure `PC`. Diversamente dalle altre regioni di memoria l'area dei metodi è trattata come un array di byte.

Per quanto riguarda i puntatori occorre precisare un aspetto. I registri `CPP`, `LV` e `SP` sono tutti puntatori a *parole*, e non a *byte*; anche i loro spiazzamenti sono espressi come numero di parole. Per il sottoinsieme d'istruzioni su interi che abbiamo scelto, tutti i riferimenti a elementi che si trovano nella porzione costante di memoria, nel blocco delle variabili locali e nello stack sono definiti come parole; allo stesso modo tutti gli spiazzamenti utilizzati all'interno di questi blocchi sono definiti in termini di parole. Per esempio `LV`, `LV + 1` e `LV + 2` fanno riferimento alle prime tre parole del blocco delle variabili locali. `LV`, `LV + 4` e `LV + 8` fanno invece riferimento a parole che si trovano a intervalli di quattro parole (16 byte) l'una dall'altra.

Al contrario, `PC` contiene un indirizzo espresso in byte; un'addizione o una sottrazione effettuata su `PC` modifica quindi l'indirizzo in base a un certo numero di byte, e non di parole. L'indirizzamento di `PC` è diverso da quello degli altri registri; la speciale porta di memoria fornita per `PC` su Mic-1 ne è una dimostrazione. Come abbiamo già visto questa porta è larga soltanto 1 byte. Incrementare `PC` di uno e dare inizio a una lettura corrisponde a prelevare il *byte* successivo; incrementare `SP` di uno e dare inizio a una lettura corrisponde invece a prelevare la *parola* successiva.

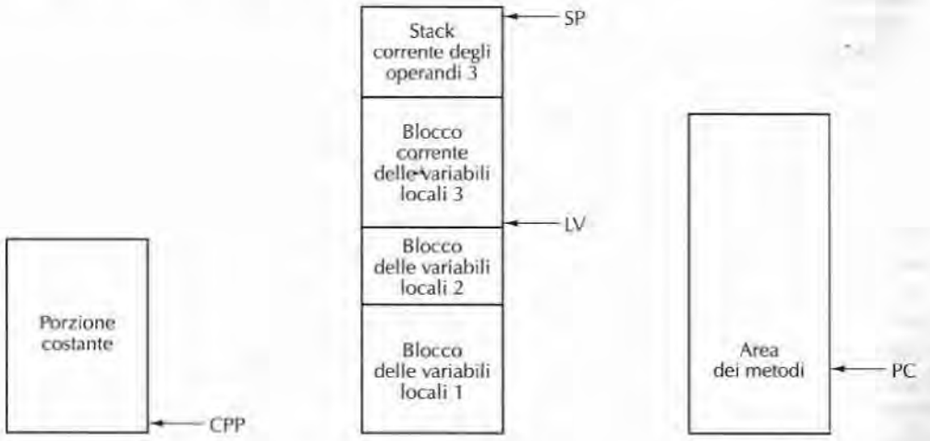


Figura 4.10 Parti della memoria di IJVM.

4.2.3 Insieme d'istruzioni IJVM

La Figura 4.11 mosta l'insieme d'istruzioni IJVM. Ogni istruzione è composta da un codice operativo e in alcuni casi da un operando, che può essere uno spiazzamento o una costante. La prima colonna mostra la codifica esadecimale dell'istruzione, la seconda il nome mnemonico in linguaggio assemblativo e la terza una breve descrizione del significato dell'istruzione.

Alcune istruzioni permettono di inserire nello stack una parola proveniente da varie fonti, come ad esempio la porzione costante di memoria (`LDC_W`), il blocco delle variabili locali (`ILOAD`) e l'istruzione stessa (`BIPUSH`). Una variabile può anche essere estratta dallo stack e memorizzata nel blocco delle variabili locali (`ISTORE`). È possibile eseguire due operazioni aritmetiche (`IADD` e `ISUB`) e due operazioni logiche, cioè booleane, (`IAND` e `IOR`) utilizzando come operandi le due parole che si trovano in cima allo stack. In tutte le operazioni logiche e aritmetiche vengono estratte due parole dallo stack e il risultato viene inserito sopra di esso. Sono fornite quattro istruzioni per i salti, una non condizionale (`GOTO`) e tre condizionali (`IFEQ`, `IFLT` e `IF_ICMPEQ`). Tutte queste istruzioni, se accettate, modificano il valore di `PC` in base alla grandezza del loro spiazzamento (16 bit con segno), che si trova nell'istruzione, subito dopo il codice operativo. Questo spiazzamento viene aggiunto all'indirizzo dell'istruzione. Ci sono anche istruzioni IJVM che permettono di scambiare le due parole in cima allo stack (`SWAP`), di duplicare la parola che si trova in cima (`DUP`) e di rimuoverla (`POP`).

Esa	Nome mnemonico	Significato
0x10	BIPUSH *byte*	Scrive un byte in cima allo stack
0x59	DUP	Legge la parola in cima allo stack e la inserisce in cima allo stack
0xA7	GOTO *offset*	Diramazione incondizionata
0x60	IADD	Sostituisce le due parole in cima allo stack con la loro somma
0x7E	IAND	Sostituisce le due parole in cima allo stack con il loro AND
0x99	IFEQ *offset*	Estrae una parola dalla cima dello stack ed effettua una diramazione se ha valore zero
0x9B	IFLT *offset*	Estrae una parola dalla cima dello stack ed effettua una diramazione se ha valore negativo
0x9F	IF_ICMPEQ *offset*	Estrae le due parole in cima allo stack ed effettua una diramazione se sono uguali
0x84	IINC *varnum const*	Aggiunge una costante a una variabile locale
0x15	ILOAD *varnum*	Scrive una variabile locale in cima allo stack
0xB6	INVOKEVIRTUAL *disp*	Invoca un metodo
0x80	IOR	Sostituisce le due parole in cima allo stack con il loro OR
0xAC	IRETURN	Termina un metodo restituendo un valore intero
0x36	ISTORE *varnum*	Preleva una parola dalla cima dello stack e la memorizza in una variabile locale
0x64	ISUB	Sostituisce le due parole in cima allo stack con la loro differenza
0x13	LDC_W *index*	Scrive in cima allo stack una costante proveniente dalla porzione costante di memoria
0x00	NOP	Non esegue nulla
0x57	POP	Rimuove la cima allo stack
0x5F	SWAP	Scambia le due parole in cima allo stack
0xC4	WIDE	Istruzione prefisso: l'istruzione successiva ha un indice a 16 bit

Figura 4.11 Insieme d'istruzioni di IJVM. Gli operandi *byte*, *const* e *varnum* sono lunghi 1 byte. Gli operandi *disp*, *index* e *offset* sono lunghi 2 byte.

Alcune istruzioni hanno più formati, per permettere di utilizzare una forma più breve per le versioni utilizzate più frequentemente. In IJVM abbiamo incluso due dei vari meccanismi che JVM mette a disposizione a questo scopo. In un caso abbiamo omesso la versione corta in favore di quella più generale. In un altro caso mostriamo come si può utilizzare il prefisso dell'istruzione `WIDE` per modificare l'istruzione successiva.

Infine c'è un'istruzione (`INVOKEVIRTUAL`) per invocare un altro metodo e un'istruzione (`IRETURN`) per terminare il metodo e restituire il controllo a quello che l'aveva invocato. A causa della complessità del meccanismo abbiamo semplificato leggermente la definizione, rendendo possibile un semplice metodo per invocare una chiamata e restituire il controllo. Questa restrizione, a differenza di quanto avviene realmente in Java, ci permette solamente di invocare un metodo che esiste all'interno dello stesso oggetto del metodo chiamante. Tale restrizione invalida severamente l'orientamento a oggetti, ma permette di non dover localizzare dinamicamente il metodo. (Se non si ha familiarità con la programmazione orientata agli oggetti si può ignorare senza alcun problema quest'ultima osservazione. Quello che abbiamo fatto è stato trasformare Java in un linguaggio non orientato agli oggetti, come C o Pascal.) Su tutti i calcolatori, fatta eccezione per la JVM, l'indirizzo della procedura da richiamare è determinato direttamente dall'istruzione `CALL`; il nostro approccio è quindi il caso normale e non rappresenta un'eccezione.

Il meccansimo per l'invocazione dei metodi funziona nel modo seguente. Prima di tutto il chiamante inserisce sullo stack un riferimento (puntatore) all'oggetto da chiamare. (Questo riferimento non è necessario in IJVM dato che non può essere specificato alcun altro oggetto, ma lo abbiamo mantenuto per coerenza con JVM.) Nella Figura 4.12(a) questo riferimento è indicato con `OBJREF`. Successivamente il chiamante inserisce sullo stack i parametri del metodo che, in questo esempio, sono *Parametro 1*, *Parametro 2* e *Parametro 3*. Infine viene eseguita l'istruzione `INVOKEVIRTUAL`.

Quest'istruzione include uno spiazzamento che indica una posizione all'interno della porzione costante di memoria; questa locazione contiene l'indirizzo dell'area dei metodi in cui inizia il metodo che si sta invocando. Tuttavia, mentre il codice del metodo risiede nella locazione puntata da questo puntatore, i primi 4 byte nell'area dei metodi contengono dati speciali. I primi 2 byte sono interpretati come un intero a 16 bit che indica il numero di parametri del metodo (i parametri stessi sono stati precedentemente inseriti in cima allo stack). In questo conteggio `OBJREF` viene considerato come un parametro: il parametro 0. Questo intero a 16 bit fornisce, insieme al valore di `SP`, la locazione di `OBJREF`. Si noti che `LV` punta a `OBJREF` invece che al primo, vero, parameto; questa scelta è sostanzialmente arbitraria.

Anche i successivi 2 byte dell'area dei metodi sono interpretati come un intero a 16 bit, che però indica la dimensione del blocco delle variabili locali per il metodo invocato.

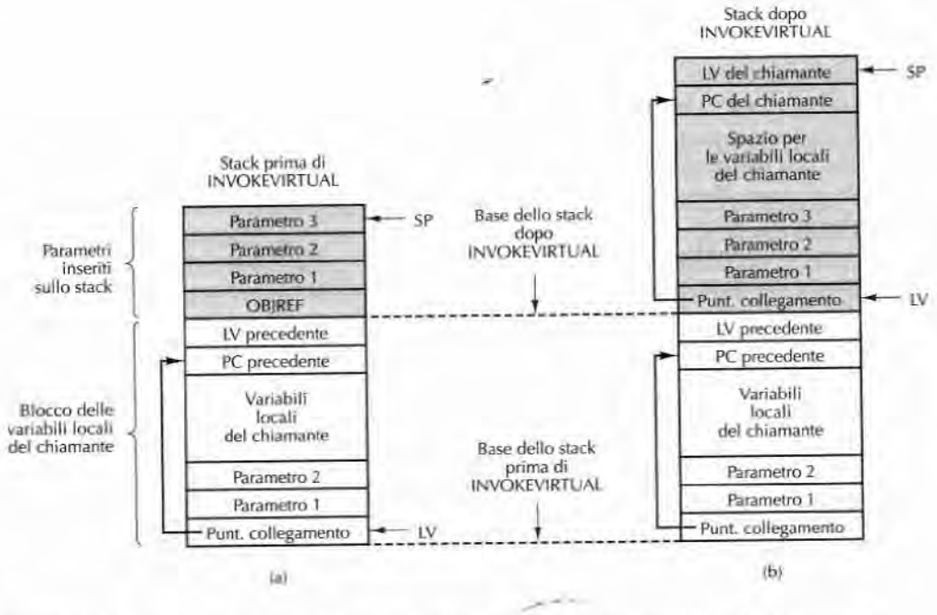


Figura 4.12 (a) Memoria prima di eseguire l'istruzione `INVOKEVIRTUAL`. (b) Dopo averla eseguita.

Questo valore è necessario, dato che verrà stabilito un nuovo stack per il metodo, immediatamente sopra il blocco delle variabili locali. Infine il quinto byte dell'area dei metodi contiene il primo codice operativo da eseguire.

Descriviamo ora l'effettiva sequenza di operazioni (Figura 4.12) che si verificano quando viene eseguita l'istruzione `INVOKEVIRTUAL`. I due byte senza segno che seguono il codice operativo sono utilizzati per costruire un indice relativo alla porzione costante di memoria (il primo byte è quello più significativo). L'istruzione calcola l'indirizzo base del nuovo blocco delle variabili locali sottraendo il numero di parametri dal puntatore allo stack e impostando `LV` in modo che punti a `OBJREF`. In questa locazione, sovrascrivendo `OBJREF`, viene memorizzato l'indirizzo della locazione in cui si trova il vecchio `PC`. Questo indirizzo è calcolato sommando la dimensione del blocco delle variabili locali (parametri + variabili locali) all'indirizzo contenuto in `LV`. Immediatamente sopra l'indirizzo in cui memorizzare il vecchio `PC` c'è l'indirizzo in cui deve essere memorizzato il vecchio `LV`. Al di sopra di questo indirizzo inizia lo stack per la procedura che è stata appena richiamata. `SP` viene impostato in modo da puntare al vecchio `LV`, che è l'indirizzo immediatamente sotto la prima locazione vuota dello stack. Ricordiamoci che `SP` punta sempre alla parola in cima allo stack. Se lo stack è vuoto, allora punta alla prima locazione sotto la fine dello stack, dato che i nostri stack crescono verso l'alto, verso gli indirizzi più grandi. Anche nelle nostre figure gli stack crescono verso l'alto, con gli indirizzi che aumentano in direzione del bordo superiore della pagina.

L'ultima operazione richiesta per terminare l'esecuzione di `INVOKEVIRTUAL` è impostare `PC` in modo che punti al quinto byte nell'area del codice del metodo.

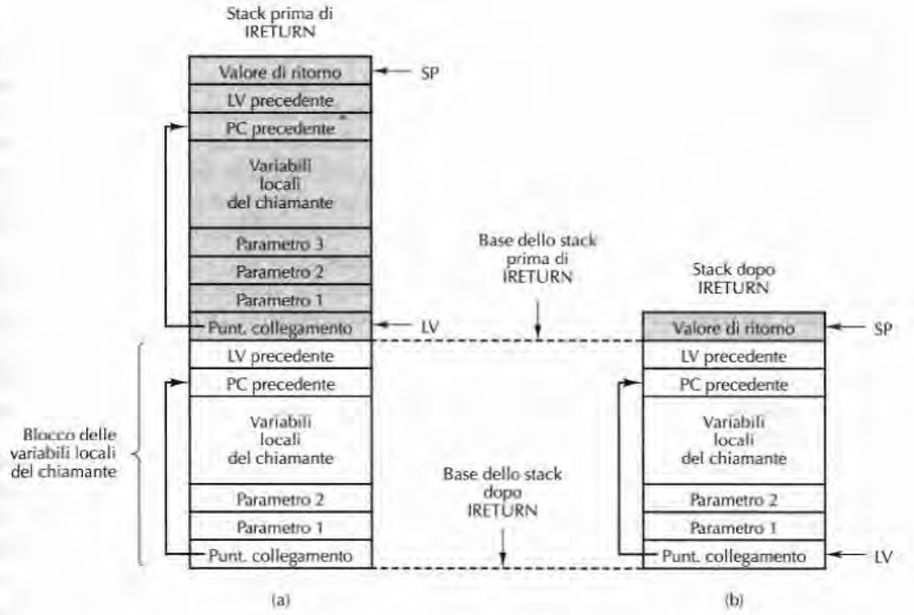


Figura 4.13 (a) Memoria prima di eseguire l'istruzione `IRETURN`. (b) Dopo averla eseguita.

L'istruzione `IRETURN` inverte la sequenza delle operazioni compiute da `INVOKEVIRTUAL`, come mostra la Figura 4.13. Essa dealloca lo spazio utilizzato dal metodo che sta restituendo il controllo e riporta lo stack nel suo precedente stato, tranne per il fatto che (1) la parola `OBJREF` (ora sovrascritta) e tutti i parametri sono stati estratti dallo stack e che (2) il valore restituito dal metodo è stato inserito in cima allo stack, nella locazione occupata precedentemente da `OBJREF`. Per memorizzare il vecchio stato l'istruzione `IRETURN` deve essere in grado di riportare i puntatori `PC` e `LV` ai loro precedenti valori. Per far ciò accede al puntatore di collegamento (*link pointer*), cioè alla parola identificata dal puntatore `LV` corrente. Non dimentichiamo che in questa locazione, in cui era precedentemente memorizzato `OBJREF`, l'istruzione `INVOKEVIRTUAL` ha memorizzato l'indirizzo contenente il vecchio `PC`. Questa parola, e quella che si trova al di sopra, vengono recuperate e utilizzate per restituire a `PC` e a `LV` i loro precedenti valori. Il valore fornito dal metodo, memorizzato in cima allo stack, viene copiato nella locazione in cui era originariamente memorizzato `OBJREF`, e `SP` viene reimpostato in modo da puntare a questa locazione. Il controllo viene quindi restituito all'istruzione immediatamente successiva rispetto all'istruzione `INVOKEVIRTUAL`.

Finora nella nostra macchina non abbiamo introdotto istruzioni di input/output, e non lo faremo neanche in seguito. Queste istruzioni non sono necessarie, a meno che non le richieda la Java Virtual Machine; tuttavia le specifiche ufficiali di JVM non fanno mai cenno alle istruzioni di I/O. La teoria è che una macchina che non esegue alcun input o output è "sicura". (In JVM lettura e scrittura sono eseguite per mezzo di chiamate a speciali metodi di I/O.)

4.2.4 Compilazione da Java a IJVM

Vediamo ora le relazioni tra Java e IJVM. Un compilatore Java che riceve in ingresso il semplice frammento di codice Java della Figura 4.14(a) dovrebbe generare qualcosa di simile al codice assemblativo della Figura 4.14(b). I commenti (preceduti dai caratteri //) e i numeri di linea alla sinistra del codice assemblativo non fanno parte dell'output del compilatore. L'assemblatore Java dovrebbe in seguito tradurre il programma assemblativo nel codice binario mostrato nella Figura 4.14(c). (In realtà il compilatore Java crea il proprio programma assemblativo e genera il codice binario direttamente.) In questo esempio abbiamo assunto che *i* sia la variabile locale 1, *j* la variabile locale 2 e *k* la variabile locale 3.

Il codice compilato è semplice. Inizialmente *j* e *k* vengono inseriti in cima allo stack, vengono sommati, e il risultato viene memorizzato in *i*. Successivamente *i* e la costante 3 vengono messi nello stack e confrontati. Se sono uguali, si effettua una diramazione che porta in *L1*, dove *k* è impostato al valore 0. Se invece sono diversi il test dà esito negativo e viene eseguito il codice che segue l'istruzione `IF_ICMPEQ`. Dopo alcune istruzioni si effettua una diramazione che porta in *L2*, cioè nel punto in cui i rami `then` e `else` si riuniscono.

La Figura 4.15 mostra come varia lo stack degli operandi durante l'esecuzione del programma IJVM della Figura 4.14(b). Prima che cominci l'esecuzione del codice lo stack è vuoto, com'è indicato dalla linea orizzontale sopra il numero 0. Dopo la prima istruzione `ILOAD`, *j* si trova sullo stack, com'è indicato dal rettangolo *j* sopra il numero 1 (questo valore significa che è stata eseguita un'istruzione). Dopo la seconda istruzione `ILOAD`, nello stack ci sono due parole, come mostrano i due rettangoli sopra il numero 2. Dopo l'istruzione `IADD`, nello stack c'è soltanto una parola, che contiene la somma *j* + *k*. Quando la parola in cima allo stack viene estratta e memorizzata in *i* lo stack è vuoto, come mostra la linea sopra il numero 4.

L'istruzione 5 (`ILOAD`) inizia il costrutto `if` inserendo *i* nello stack (in 5). Successivamente (in 6) arriva la costante 3. Dopo il confronto lo stack è nuovamente vuoto (7). L'istruzione 8 rappresenta l'inizio del ramo `else` del frammento di programma Java. La parte `else` continua fino all'istruzione 12, punto in cui c'è la diramazione che porta a *L2* e che permette di saltare il codice relativo al ramo `then`.

```
i = j + k;                 1        ILOAD j        // i = j + k      0x15 0x02
if (i == 3)                2        ILOAD k                       0x15 0x03
   k = 0;                  3        IADD                          0x60
else                       4        ISTORE i                      0x36 0x01
   j = j - 1;              5        ILOAD i        // if (i == 3)   0x15 0x01
                           6        BIPUSH 3                      0x10 0x03
                           7        IF_ICMPEQ L1                  0x9F 0x00 0x0D
                           8        ILOAD j        // j = j - 1     0x15 0x02
                           9        BIPUSH 1                      0x10 0x01
                          10        ISUB                          0x64
                          11        ISTORE j                      0x36 0x02
                          12        GOTO L2                       0xA7 0x00 0x07
                          13   L1:  BIPUSH 0       // k = 0         0x10 0x00
                          14        ISTORE k                      0x36 0x03
                          15   L2:
        (a)                              (b)                            (c)
```

Figura 4.14 (a) Frammento di programma Java. (b) Il frammento in linguaggio assemblativo Java. (c) Programma IJVM in esadecimale.

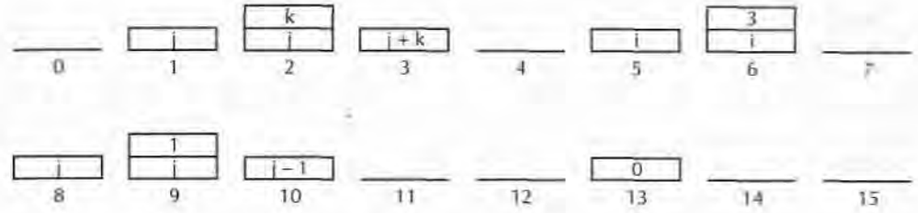


Figura 4.15 Stack dopo ogni istruzione della Figura 4.14(b).

4.3 Implementazione di esempio

Dopo aver specificato nel dettaglio la microarchitettura e la macroarchitettura, rimane il problema dell'implementazione. In altre parole, che aspetto ha e come funziona un programma che viene eseguito sulla prima e che interpreta la seconda? Prima di poter rispondere a queste domande dobbiamo considerare in modo preciso la notazione che utilizzeremo per descrivere l'implementazione.

4.3.1 Microistruzioni e notazione

In teoria potremmo descrivere la memoria di controllo in binario, 36 bit per parola. Ma, come avviene normalmente nei linguaggi di programmazione, è molto vantaggioso introdurre una notazione che esprime l'essenza dei problemi da trattare mascherando allo stesso tempo i dettagli che possono essere ignorati o che possono essere meglio gestiti in maniera automatica. È importante capire che il linguaggio da noi scelto ha lo scopo di illustrare i concetti e non di facilitare la progettazione di un'architettura efficiente. Se questo fosse il nostro obiettivo dovremmo utilizzare una differente notazione, in grado di fornire al progettista la massima flessibilità possibile. Un aspetto in cui questo problema è rilevante riguarda la scelta degli indirizzi.

Dato che la memoria non è ordinata dal punto di vista logico, non esiste alcuna naturale "istruzione successiva" da poter impiegare quando specifichiamo una sequenza di operazioni. Molta della potenza dell'organizzazione del controllo dipende dalla capacità del progettista (o dell'assemblatore) di specificare gli indirizzi in modo efficiente. Cominciamo dunque con l'introdurre un semplice linguaggio simbolico che descrive in modo completo le operazioni senza spiegare in modo esaustivo come sono stati determinati tutti gli indirizzi.

La nostra notazione specifica su una singola linea tutte le attività che si svolgono durante un unico ciclo di clock. In teoria per descrivere le operazioni potremmo utilizzare un linguaggio ad alto livello. Tuttavia un controllo ciclo-dopo-ciclo è molto importante, in quanto dà la possibilità di effettuare più operazioni concorrenti; inoltre è necessario per poter analizzare i cicli in modo da comprenderne lo svolgimento e verificare le operazioni. Se l'obiettivo è quello di definire un'implementazione veloce ed efficiente (a parità di altri fattori, veloce ed efficiente è sempre meglio che lento e inefficiente) ogni ciclo ha la sua importanza. Le implementazioni reali celano molte sottili astuzie, come l'uso di sequenze o operazioni difficilmente comprensibili per cercare di risparmiare anche un solo ciclo. Risparmiare

cicli ha un importante impatto sulle prestazioni: se un'istruzione a quattro cicli può essere ridotta a due cicli, allora verrà eseguita al doppio della velocità.

Un possibile approccio per definire la notazione si limita all'elenco dei segnali che dovrebbero essere attivati durante ciascun ciclo di clock. Supponiamo che in un ciclo vogliamo incrementare il valore di SP, oltre a dare inizio a un'operazione di lettura; desideriamo inoltre che l'istruzione successiva sia quella che risiede nella locazione 122 della memoria di controllo. Potremmo scrivere

```
ReadRegister = SP, ALU = INC, WSP, Read, NEXT_ADDRESS = 122
```

dove WSP significa "scrivere il registro SP". Questa notazione è completa, ma di difficile lettura. Al suo posto preferiamo combinare le operazioni in un modo naturale e intuitivo in modo da cogliere la sostanza di quello che sta avvenendo.

```
SP = SP + 1; rd
```

Chiamiamo il nostro linguaggio ad alto livello Micro Assembly Language, "MAL" (iniziale di "malato", quello che uno diventa se è obbligato a scrivere molte linee di codice in questo linguaggio). MAL è definito appositamente per rispecchiare le caratteristiche della microarchitettura. Durante ogni ciclo è possibile scrivere qualsiasi registro, anche se in genere ne viene scritto soltanto uno. Soltanto un registro può essere collegato al lato B della ALU, mentre sul lato A le scelte possibili sono +1, 0, –1 e il registro H. Possiamo inoltre usare un semplice costrutto di assegnamento, come in Java, per indicare l'operazione da eseguire; per copiare un dato da SP a MDR possiamo per esempio dire

```
MDR = SP
```

Per indicare che si intende utilizzare funzioni della ALU diverse dal semplice passaggio verso il bus B possiamo scrivere

```
MDR = H + SP
```

che somma il contenuto del registro H a SP e scrive il risultato in MDR. L'operatore + è commutativo (il che significa che l'ordine degli operandi è ininfluente); quindi l'istruzione può essere riscritta come

```
MDR = SP + H
```

e genera la stessa microistruzione a 36 bit; per essere precisi H deve però essere l'operando sinistro della ALU.

Dobbiamo fare attenzione a usare soltanto le operazioni lecite. Quelle più importanti sono mostrate nella Figura 4.16, in cui SOURCE può essere MDR, PC, MBR, MBRU, SP, LV, CPP, TOS oppure OPC (MBRU indica la versione senza segno di MBR). Tutti questi registri possono fungere da sorgenti per la ALU sul bus B. In modo analogo DEST può essere uno qualsiasi fra MAR, MDR, PC, SP, LV, CPP, TOS, OPC e H; tutti questi registri possono essere la destinazione dell'output della ALU che viene inviato sul bus C. Questo formato è insidioso, in quanto molti costrutti apparentemente corretti sono invece illegali. Per esempio, l'assegnamento

```
MDR = SP + MDR
```

sembra perfettamente ragionevole, ma non può in alcun modo essere eseguito in un unico ciclo sul percorso dati della Figura 4.6. Questa restrizione è dovuta al fatto che in un'addi-

DEST = H
DEST = SOURCE
DEST = $\overline{H}$
DEST = $\overline{SOURCE}$
DEST = H + SOURCE
DEST = H + SOURCE + 1
DEST = H + 1
DEST = SOURCE + 1
DEST = SOURCE – H
DEST = SOURCE – 1
DEST = –H
DEST = H AND SOURCE
DEST = H OR SOURCE
DEST = 0
DEST = 1
DEST = –1

Figura 4.16 Operazioni consentite. Possono tutte essere estese aggiungendo "<< 8" per traslare il risultato a sinistra di 1 byte. Un'operazione comune è H = MBR << 8.

zione (fatta eccezione per un incremento o un decremento) uno degli operandi deve essere il registro H. Analogamente, l'assegnamento

```
H = H – MDR
```

potrebbe essere utile, ma è anch'esso ineseguibile dato che l'unica possibile sorgente per il sottraendo è il registro H. È compito dell'assemblatore rifiutare quelle istruzioni che sembrano valide, ma in realtà non lo sono.

Estendiamo ora la notazione per permettere assegnamenti multipli mediante la ripetizione del simbolo di uguaglianza. Per esempio, per sommare 1 a SP, memorizzare il risultato nuovamente in SP e scriverlo anche all'interno di MDR, si può utilizzare la seguente notazione

```
SP = MDR = SP + 1
```

Per indicare letture e scritture di parole di dati da 4 byte aggiungeremo semplicemente rd e wr nella microistruzione. Il prelievo di un byte attraverso la porta a 1 byte si indica con fetch. Gli assegnamenti e le operazioni della memoria possono svolgersi durante lo stesso ciclo, il che è indicato scrivendoli sulla stessa linea.

Per evitare possibili confusioni ripetiamo che Mic-1 ha due modi per accedere alla memoria. Le letture e le scritture di parole a 4 byte usano MAR/MDR e nelle microistruzioni sono indicate rispettivamente da rd e wr. Le letture dei codici operativi a 1 byte dal flusso dell'istruzione utilizzano PC/MBR e sono indicate nelle microistruzioni mediante la parola fetch. Le due operazioni possono procedere in modo simultaneo.

Tuttavia lo stesso registro non può ricevere un valore dalla memoria e dal percorso dati durante lo stesso ciclo. Consideriamo il codice

```
MAR = SP; rd
MDR = H
```

La prima microistruzione assegna alla fine della seconda microistruzione un valore della memoria a MDR. Tuttavia anche la seconda microistruzione assegna nello stesso momento un valore a MDR. Questi due assegnamenti sono in conflitto e non sono permessi, poiché in tal caso il risultato sarebbe indefinito.

Ricordiamoci che ogni microistruzione deve fornire in modo esplicito l'indirizzo della successiva microistruzione da eseguire. Tuttavia spesso succede che una microistruzione sia invocata solamente da un'altra microistruzione, in particolare da quella che si trova nella linea immediatamente sopra di essa. Per facilitare il lavoro del programmatore normalmente il microassemblatore assegna un indirizzo a ciascuna microistruzione (non necessariamente consecutiva all'interno della memoria di controllo) e completa il campo NEXT_ADDRESS in modo che le microistruzioni scritte su linee consecutive vengano eseguite in sequenza.

Tuttavia in alcuni casi il microprogrammatore desidera inserire una diramazione, condizionale oppure no. La notazione per i salti incondizionati è semplice:

```
goto label
```

e può essere inclusa in ogni microistruzione per indicare in modo esplicito il proprio successore. Per esempio, la maggior parte delle sequenze di microistruzioni termina con un ritorno alla prima istruzione del ciclo principale. Dunque l'ultima istruzione di queste sequenze include generalmente

```
goto Main1
```

Si noti che anche quando una microistruzione contiene goto il percorso dati è disponibile per le normali operazioni. Dopo tutto ogni microistruzione contiene un campo NEXT_ADDRESS e quello che fa goto non è altro che comunicare al microassemblatore di inserire in questo campo un particolare valore, al posto dell'indirizzo in cui esso aveva deciso di mettere la microistruzione della linea successiva. In linea di principio ogni linea dovrebbe avere un'istruzione goto che però, per comodità, viene omessa quando l'indirizzo di destinazione è la linea successiva.

Per i salti condizionali abbiamo invece bisogno di una notazione differente. Ricordiamoci che JAMN e JAMZ usano i bit N e Z, che vengono impostati in base all'output della ALU. A volte è necessario testare un registro per vedere se il suo valore è 0. Un modo per farlo consiste nel far passare il valore del registro attraverso la ALU e nel memorizzare il risultato nuovamente nel registro stesso. Scrivere

```
TOS = TOS
```

può sembrare strano, ma svolge correttamente il compito (impostando il flip-flop Z in base a TOS). Tuttavia per migliorare l'aspetto dei microprogrammi estendiamo MAL, aggiungendo due nuovi registri immaginari, N e Z, a cui è possibile effettuare degli assegnamenti. Per esempio,

```
Z = TOS
```

fa passare TOS attraverso la ALU, impostando quindi i flip-flop Z (e N), senza però effettuare alcuna memorizzazione nei registri. Utilizzare Z oppure N come destinazione non fa altro che indicare al microassemblatore di impostare a 0 tutti i bit del campo C della Figura 4.5. Il percorso dati esegue un ciclo in cui sono consentite tutte le normali operazioni, ma in cui non viene scritto alcun registro. Si noti che è ininfluente il fatto che la destinazione sia N oppure Z, dato che la microistruzione generata dal microassemblatore è identica. I programmatori che scelgono intenzionalmente quella "sbagliata", per punizione dovrebbero essere obbligati a lavorare per una settimana intera con il PC IBM originale a 4,77 MHz!

La sintassi per comunicare al microassemblatore di impostare il bit JAMZ è

```
if (Z) goto L1; else goto L2
```

Dato che l'hardware richiede che gli 8 bit meno significativi di questi indirizzi siano identici, è compito del microassemblatore assegnare gli indirizzi correttamente. D'altra parte, dato che *L2* può trovarsi in un qualsiasi punto delle prime 256 parole della memoria di controllo, il microassemblatore ha molta libertà nel cercare un altro indirizzo che possa essere accoppiato al primo.

In genere queste ultime due istruzioni vengono combinate; per esempio

```
Z = TOS; if (Z) goto L1; else goto L2
```

Come risultato MAL genera una microistruzione in cui TOS viene fatto passare attraverso la ALU (ma senza essere memorizzato in alcun registro) in modo che il suo valore imposti il bit Z. Poco dopo aver caricato in Z il bit di condizione della ALU, viene calcolato in OR all'interno del bit più significativo di MPC, obbligando a prelevare l'indirizzo della successiva microistruzione o da *L2* oppure da *L1* (che deve essere 256 più di *L2*). MPC sarà stabile e pronto per essere utilizzato nella successiva microistruzione.

Infine abbiamo bisogno di una notazione per utilizzare il bit JMPC. Quella che adotteremo è

```
goto (MBR OR valore)
```

Questa sintassi indica al microassemblatore di utilizzare *valore* per NEXT_ADDRESS e di impostare il bit JMPC in modo che in MPC venga calcolato l'OR logico tra MBR e NEXT_ADDRESS. Se *valore* è 0, allora la situazione è quella normale ed è sufficiente scrivere semplicemente

```
goto (MBR)
```

Si noti che in questo caso non si verifica il problema dell'estensione del segno (cioè la differenza tra MBR e MBRU), dato che soltanto gli 8 bit meno significativi di MBR sono collegati a MPC (vedi Figura 4.6). Si osservi inoltre che quello che si utilizza è il valore di MBR disponibile alla fine del ciclo corrente. Un prelievo che inizia in *questa* microistruzione avviene troppo in ritardo per poter modificare la scelta della micoistruzione successiva.

4.3.2 Implementazione di IJVM con Mic-1

Siamo finalmente arrivati al momento in cui possiamo mettere insieme tutti i pezzi. La Figura 4.17 è il microprogramma eseguito su Mic-1 che interpreta IJVM. È un programma sorprendentemente corto, con un totale di sole 112 microistruzioni. Per ciascuna microistruzione ci sono tre colonne: l'etichetta simbolica, il microcodice effettivo e un commento. Come abbiamo già sottolineato più volte, si può notare che microistruzioni consecutive non sono necessariamente localizzate in indirizzi consecutivi all'interno della memoria di controllo.

Etichetta	Operazioni	Commenti
Main1	PC = PC + 1; fetch; goto (MBR)	MBR contiene il codice operativo; prelievo del byte successivo; diramazione
nop1	goto Main1	Non esegue nulla
iadd1	MAR = SP = SP – 1; rd	Legge la seconda parola in cima allo stack
iadd2	H = TOS	H = cima dello stack
iadd3	MDR = TOS = MDR + H; wr; goto Main1	Somma le due parole in cima allo stack; scrive in cima allo stack
isub1	MAR = SP = SP – 1; rd	Legge la seconda parola in cima allo stack
isub2	H = TOS	H = cima dello stack
isub3	MDR = TOS = MDR – H; wr; goto Main1	Esegue la sottrazione; scrive in cima allo stack
iand1	MAR = SP = SP – 1; rd	Legge la seconda parola in cima allo stack
iand2	H = TOS	H = cima dello stack
iand3	MDR = TOS = MDR AND H; wr; goto Main1	Esegue l'AND; scrive nella nuova cima dello stack
ior1	MAR = SP = SP – 1; rd	Legge la seconda parola in cima allo stack
ior2	H = TOS	H = cima dello stack
ior3	MDR = TOS = MDR OR H; wr; goto Main1	Esegue l'OR; scrive nella nuova cima dello stack
dup1	MAR = SP = SP + 1	Incrementa SP e lo copia in MAR
dup2	MDR = TOS; wr; goto Main1	Scrive la nuova parola dello stack
pop1	MAR = SP = SP – 1; rd	Legge la seconda parola in cima allo stack
pop2		Attende che il nuovo TOS sia letto dalla memoria
pop3	TOS = MDR; goto Main1	Copia la nuova parola in TOS
swap1	MAR = SP – 1; rd	Imposta MAR a SP – 1; legge la seconda parola dallo stack
swap2	MAR = SP	Imposta MAR con la parola in cima allo stack
swap3	H = MDR; wr	Salva TOS in H; scrive la seconda parola in cima allo stack
swap4	MDR = TOS	Copia il vecchio TOS in MDR
swap5	MAR = SP – 1; wr	Imposta MAR a SP – 1; scrive la seconda parola nello stack
swap6	TOS = H; goto Main1	Aggiorna TOS
bipush1	SP = MAR = SP + 1	MBR = byte da inserire nello stack
bipush2	PC = PC + 1; fetch	Incrementa PC, preleva il successivo codice operativo
bipush3	MDR = TOS = MBR; wr; goto Main1	Estende il segno della costante e la inserisce nello stack
iload1	H = LV	MBR contiene l'indice; copia LV in H
iload2	MAR = MBRU + H; rd	MAR = indirizzo della variabile locale da inserire nello stack
iload3	MAR = SP = SP + 1	SP punta alla nuova cima dello stack; prepara la scrittura
iload4	PC = PC + 1; fetch; wr	Incrementa PC; ottiene il nuovo codice operativo; scrive la cima dello stack
iload5	TOS = MDR; goto Main1	Aggiorna TOS
istore1	H = LV	MBR contiene l'indice; copia LV in H
istore2	MAR = MBRU + H	MAR = indirizzo della variabile locale da memorizzare
istore3	MDR = TOS; wr	Copia TOS in MDR; scrive la parola
istore4	SP = MAR = SP – 1; rd	Legge la nuova parola in cima allo stack
istore5	PC = PC + 1; fetch	Incrementa PC; preleva il successivo codice operativo
istore6	TOS = MDR; goto Main1	Aggiorna TOS

Figura 4.17 Microprogramma per Mic-1.

Etichetta	Operazioni	Commenti
wide1	PC = PC + 1; fetch;	Preleva il byte dell'operando o il successivo codice operativo
wide2	goto (MBR OR 0x100)	Diramazione a più destinazioni con il bit alto impostato a 1
wide_iload1	PC = PC + 1; fetch	MBR contiene il primo byte dell'indice; preleva il secondo
wide_iload2	H = MBRU << 8	H = primo byte dell'indice traslato a sinistra di 8 bit
wide_iload3	H = MBRU OR H	H = indice a 16 bit della variabile locale
wide_iload4	MAR = LV + H; rd; goto iload3	MAR = indirizzo della variabile locale da inserire nello stack
wide_istore1	PC = PC + 1; fetch	MBR contiene il primo byte dell'indice; preleva il secondo
wide_istore2	H = MBRU << 8	H = primo byte dell'indice traslato a sinistra di 8 bit
wide_istore3	H = MBRU OR H	H = indice a 16 bit della variabile locale
wide_istore4	MAR = LV + H; goto istore3	MAR = indirizzo della variabile locale da memorizzare
ldc_w1	PC = PC + 1; fetch	MBR contiene il primo byte dell'indice; preleva il secondo
ldc_w2	H = MBRU << 8	H = primo byte dell'indice << 8
ldc_w3	H = MBRU OR H	H = indice a 16 bit relativo alla porzione costante di memoria
ldc_w4	MAR = H + CPP; rd; goto iload3	MAR = indirizzo della costante nella porzione costante
iinc1	H = LV	MBR contiene l'indice; copia LV in H
iinc2	MAR = MBRU + H; rd	Copia LV + indice in MAR; legge la variabile
iinc3	PC = PC + 1; fetch	Preleva la costante
iinc4	H = MDR	Copia la variabile in H
iinc5	PC = PC + 1; fetch	Preleva il successivo codice operativo
iinc6	MDR = MBR + H; wr; goto Main1	Inserisce in MDR la somma; aggiorna la variabile
goto1	OPC = PC – 1	Salva l'indirizzo del codice operativo
goto2	PC = PC + 1; fetch	MBR contiene il primo byte dell'indice; preleva il secondo
goto3	H = MBR << 8	Trasla e salva in H il primo byte con segno
goto4	H = MBRU OR H	H = spiazzamento a 16 bit della diramazione
goto5	PC = OPC + H; fetch	Aggiunge lo spiazzamento a OPC
goto6	goto Main1	Attende il prelievo del successivo codice operativo
iflt1	MAR = SP = SP – 1; rd	Legge la seconda parola in cima allo stack
iflt2	OPC = TOS	Salva temporaneamente TOS in OPC
iflt3	TOS = MDR	Inserisce in TOS la nuova cima dello stack
iflt4	N = OPC; if (N) goto T; else goto F	Diramazione in base al bit N
ifeq1	MAR = SP = SP – 1; rd	Legge la seconda parola in cima allo stack
ifeq2	OPC = TOS	Salva temporaneamente TOS in OPC
ifeq3	TOS = MDR	Inserisce in TOS la nuova cima dello stack
ifeq4	Z = OPC; if (Z) goto T; else goto F	Diramazione in base al bit Z
if_icmpeq1	MAR = SP = SP – 1; rd	Legge la seconda parola in cima allo stack
if_icmpeq2	MAR = SP = SP – 1	Imposta MAR per leggere la nuova cima dello stack
if_icmpeq3	H = MDR; rd	Copia in H la seconda parola dello stack
if_icmpeq4	OPC = TOS	Salva temporaneamente TOS in OPC
if_icmpeq5	TOS = MDR	Inserisce in TOS la nuova cima dello stack
if_icmpeq6	Z = OPC – H; if (Z) goto T; else goto F	Se le due parole in cima allo stack sono uguali, goto T, altrimenti, goto F
T	OPC = PC – 1; goto goto2	Uguale a goto1; necessario per l'indirizzo destinazione

Figura 4.17 Microprogramma per Mic-1 (continua).

Etichetta	Operazioni	Commenti
F	PC = PC + 1	Salta il primo byte dello spiazzamento
F2	PC = PC + 1; fetch	PC punta ora al nuovo codice operativo
F3	goto Main1	Attende il prelievo del codice operativo
invokevirtual1	PC = PC + 1; fetch	MBR = byte 1 dell'indice; incrementa PC; ottiene il secondo byte
invokevirtual2	H = MBRU << 8	Trasla e salva in H il primo byte
invokevirtual3	H = MBRU OR H	H = spiazzamento del puntatore al metodo rispetto a CPP
invokevirtual4	MAR = CPP + H; rd	Ottiene dall'area CPP un puntatore al metodo
invokevirtual5	OPC = PC + 1	Salva temporaneamente il PC di ritorno in OPC
invokevirtual6	PC = MDR; fetch	PC punta ora al nuovo metodo; ottiene il numero di parametri
invokevirtual7	PC = PC + 1; fetch	Preleva il secondo byte del numero di parametri
invokevirtual8	H = MBRU << 8	Trasla e salva in H il primo byte
invokevirtual9	H = MBRU OR H	H = numero di parametri
invokevirtual10	PC = PC + 1; fetch	Preleva il primo byte del numero di variabili locali
invokevirtual11	TOS = SP – H	TOS = indirizzo di OBJREF – 1
invokevirtual12	TOS = MAR = TOS + 1	TOS = indirizzo di OBJREF (nuovo LV)
invokevirtual13	PC = PC + 1; fetch	Preleva il secondo byte del numero di variabili locali
invokevirtual14	H = MBRU << 8	Trasla e salva in H il primo byte
invokevirtual15	H = MBRU OR H	H = numero di variabili locali
invokevirtual16	MDR = SP + H + 1; wr	Sovrascrive OBJREF con il puntatore di collegamento
invokevirtual17	MAR = SP = MDR;	Imposta SP e MAR con la locazione in cui memorizzare il vecchio PC
invokevirtual18	MDR = OPC; wr S	Salva il vecchio PC sopra le variabili locali
invokevirtual19	MAR = SP = SP + 1	SP punta alla locazione in cui salvare il vecchio LV
invokevirtual20	MDR = LV; wr	Salva il vecchio LV sopra il PC salvato
invokevirtual21	PC = PC + 1; fetch	Preleva il primo codice operativo del nuovo metodo
invokevirtual22	LV = TOS; goto Main1	Imposta LV per puntare al blocco LV
ireturn1	MAR = SP = LV; rd	Reimposta SP e MAR per ricevere il puntatore di collegamento
ireturn2		Attende che si completi la lettura
ireturn3	LV = MAR = MDR; rd	Imposta LV con il puntatore di collegamento; ottiene il vecchio PC
ireturn4	MAR = LV + 1	Imposta MAR per leggere il vecchio LV
ireturn5	PC = MDR; rd; fetch	Ripristina PC; preleva il successivo codice operativo
ireturn6	MAR = SP	Imposta MAR per scrivere TOS
ireturn7	LV = MDR	Ripristina LV
ireturn8	MDR = TOS; wr; goto Main1	Salva il valore di ritorno sulla cima originale dello stack

Figura 4.17 Microprogramma per Mic-1 (continua).

Oramai le scelte dei nomi per la maggior parte dei registri della Figura 4.1 dovrebbero essere ovvie: CPP, LV e SP sono utilizzati per memorizzare, rispettivamente, i puntatori alla porzione costante di memoria, al blocco delle variabili locali e alla cima dello stack, mentre PC mantiene l'indirizzo del successivo byte da prelevare dal flusso dell'istruzione. MBR è un registro a 1 byte che mantiene in modo sequenziale i byte dello stream dell'istruzione provenienti dalla memoria per poterli interpretare. TOS e OPC sono registri aggiuntivi il cui utilizzo verrà descritto in seguito.

In qualsiasi momento ognuno di questi registri contiene un particolare valore; se necessario è tuttavia possibile utilizzare ciascuno di loro come un registro temporaneo. All'inizio e alla fine di ogni istruzione, TOS contiene il valore puntato da SP, la parola che si trova in cima allo stack. Questo valore è ridondante, in quanto la parola può essere letta in qualsiasi momento dalla memoria; ciononostante avendola a disposizione in un registro è spesso possibile risparmiare un riferimento alla memoria. Per poche istruzioni il mantenimento di TOS

implica un numero maggiore di operazioni di memoria. Per esempio, l'istruzione POP deve leggere dalla memoria la nuova parola che si trova in cima allo stack per poterla copiare all'interno di TOS.

Il registro OPC è un registro temporaneo (cioè di "appunti") e non ha un utilizzo predefinito. È usato per esempio per salvare l'indirizzo del codice operativo di un'istruzione di diramazione, mentre PC viene incrementato per accedere ai parametri. Viene utilizzato come registro temporaneo anche nelle istruzioni di diramazione condizionale di IJVM.

Come tutti gli interpreti, il microprogramma della Figura 4.17 ha un ciclo principale che preleva, decodifica ed esegue le istruzioni del programma interpretato, in questo caso le istruzioni IJVM. Il suo ciclo principale inizia nella linea etichettata con Main1. All'inizio del ciclo deve valere la condizione che PC sia già stato caricato con un indirizzo di una locazione di memoria contenente un codice operativo; inoltre questo codice operativo deve già essere stato memorizzato all'interno di MBR. Ciò implica che prima di iniziare ogni iterazione dobbiamo assicurarci che PC sia stato aggiornato in modo da puntare al successivo codice operativo da interpretare e che il byte del codice operativo stesso sia già stato portato all'interno di MBR.

All'inizio di ogni istruzione viene eseguita la stessa sequenza di operazioni; è importante che la lunghezza di questa sequenza sia la più breve possibile. Attraverso un'attenta progettazione dell'hardware Mic-1 e del software siamo riusciti a ridurre il ciclo principale a una sola istruzione. A partire dal momento in cui viene avviata la macchina, ogni volta che viene eseguita questa microistruzione il codice operativo IJVM da eseguire si trova già all'interno in MBR. Quello che fa la microistruzione è effettuare un salto nel microcodice per eseguire questa istruzione IJVM, oltre a iniziare il prelievo del byte che segue il codice operativo; questo byte potrebbe essere un operando oppure un nuovo codice operativo.

A questo punto possiamo svelare il vero motivo per cui le istruzioni non vengono eseguite sequenzialmente, ma ciascuna di loro è obbligata a indicare esplicitamente il proprio successore. Tutti gli indirizzi della memoria di controllo corrispondenti ai codici operativi devono essere riservati per la prima parola della corrispondente istruzione dell'interprete. Dalla Figura 4.11 vediamo quindi che il codice che interpreta POP inizia in 0x57, mentre il codice che interpreta DUP inizia in 0x59. (Come fa MAL a sapere che POP deve iniziare in 0x57 è uno dei misteri dell'universo, probabilmente esiste da qualche parte un documento segreto che ne spiega il motivo.)

Sfortunatamente il codice corrispondente a POP è lungo tre microistruzioni e quindi, se fosse memorizzato in parole consecutive, interferirebbe con l'inizio di DUP. All'interno di ciascuna sequenza le microistruzioni che seguono quella iniziale devono essere memorizzate nelle locazioni di memoria ancora libere, non riservate per i codici operativi. Per questo motivo occorre effettuare molti salti da una locazione di memoria all'altra; se regolarmente, dopo poche microistruzoni, occorresse utilizzare microdiramazioni esplicite (cioè microistruzioni che effettuano salti) per saltare da un buco a un altro della memoria, il tempo sprecato sarebbe significativo.

Per vedere come funziona l'interprete, assumiamo che MBR contenga il valore 0x60, cioè il codice operativo corrispondente a IADD (vedi Figura 4.11). Nel ciclo principale composto da una sola microistruzione compiamo tre azioni.

1. Incrementiamo PC, in modo che contenga l'indirizzo del primo byte dopo il codice operativo.

2. Iniziamo a prelevare il byte successivo da portare all'interno di MBR. Questo byte, prima o poi, si rivelerà necessario, o come operando per l'istruzione IJVM corrente oppure come codice operativo successivo (come nel caso dell'istruzione IADD, che non ha il byte per l'operando).
3. All'inizio di Main1 effettuiamo una diramazione verso l'indirizzo contenuto in MBR. Questo indirizzo, memorizzato nel registro dalla precedente microistruzione, equivale al valore numerico del codice operativo che in quel momento è in esecuzione. Occorre prestare particolare attenzione al fatto che il valore che si sta estraendo in questa microistruzione non gioca alcun ruolo nella diramazione.

Il prelievo del byte successivo comincia in questo punto in modo che sia disponibile all'inizio della terza microistruzione. In seguito potrebbe risultare necessario oppure no, ma conviene comunque far partire il prelievo, dato che la cosa non produce assolutamente alcun danno.

Se il byte contenuto in MBR è composto da soli 0, allora identifica il codice operativo di un'istruzione NOP. In questo caso la successiva microistruzione viene prelevata dalla locazione 0 ed è quella etichettata con nop1. Dato che questa istruzione non esegue alcunché essa effettua semplicemente un salto all'inizio del ciclo principale, dove viene ripetuta la sequenza utilizzando però il nuovo codice operativo memorizzato in MBR.

Sottolineiamo ancora una volta il fatto che nella Figura 4.17 le microistruzioni non sono memorizzate consecutivamente all'interno della memoria e che Main1 non si trova all'indirizzo 0 della memoria di controllo (dato che all'indirizzo 0 deve essere memorizzata nop1). È compito del microassemblatore posizionare le microistruzioni negli indirizzi adatti e collegarle in sequenze di breve lunghezza usando il campo NEXT_ADDRESS. Ogni sequenza inizia all'indirizzo corrispondente al valore numerico del codice operativo IJVM che interpreta (POP comincia per esempio in 0x57), ma il resto della sequenza può trovarsi in qualsiasi punto della memoria di controllo, e non necessariamente negli indirizzi immediatamente successivi.

Consideriamo ora l'istruzione IADD di IJVM. Dopo la diramazione presente nel ciclo principale si raggiunge la microistruzione con l'etichetta iadd1. Questa istruzione dà inizio alle operazioni specifiche di IADD.

1. TOS è già presente, mentre occorre prelevare dalla memoria la seconda parola a partire dalla cima dello stack.
2. TOS deve essere sommata alla parola che è appena stata prelevata dalla memoria.
3. Il risultato, che deve essere inserito in cima allo stack, va memorizzato sia in memoria sia nel registro TOS.

Per poter prelevare l'operando dalla memoria è necessario decrementare il puntatore allo stack e scriverlo all'interno di MAR. Si noti che, per praticità, questo indirizzo è anche l'indirizzo che sarà utilizzato nella successiva scrittura. Inoltre, dato che questa locazione sarà la nuova cima dello stack, occorre assegnare questo valore anche a SP. Una singola operazione può quindi determinare il nuovo valore di SP e MAR, decrementare SP e scriverlo in entrambi i registri.

Queste azioni vengono svolte durante il primo ciclo, iadd1, mentre inizia l'operazione di lettura. Inoltre MPC ottiene l'indirizzo di iadd2 leggendo il valore del campo

NEXT_ADDRESS di iadd1. Successivamente iadd2 viene letta dalla memoria di controllo. Durante il secondo ciclo, nell'attesa che l'operando sia letto dalla memoria, la parola in cima allo stack viene copiata da TOS a H, in modo che, una volta completata la lettura, sia disponibile per il calcolo dell'addizione.

All'inizio del terzo ciclo, iadd3, MDR contiene l'addendo prelevato dalla memoria. In questo ciclo l'addendo viene sommato al contenuto di H, e il risultato viene memorizzato sia all'interno di MDR sia all'interno di TOS. Viene fatta partire anche un'operazione di scrittura per memorizzare all'interno della memoria la nuova parola che si trova in cima allo stack. In questo ciclo goto assegna a MPC l'indirizzo di Main1, facendoci tornare al punto d'inizio, in modo da poter eseguire l'istruzione seguente.

Se il successivo codice operativo IJVM contenuto in MBR è 0x64 (ISUB), la sequenza di eventi che si svolge è praticamente identica a quella appena vista. Dopo aver eseguito Main1, il controllo viene trasferito alla microistruzione che si trova all'indirizzo 0x64 (isub1). Questa microistruzione è seguita da isub2 e isub3, e poi nuovamente da Main1. L'unica differenza tra questa sequenza e la precedente è che in isub3 il contenuto di H viene sottratto da MDR invece di sommarlo.

L'interpretazione di IAND è praticamente identica a quella di IADD e di ISUB, tranne per il fatto che si calcola l'AND tra le due parole in cima allo stack invece di sommarle oppure sottrarle. Nel caso dell'istruzione IOR la situazione è del tutto simile.

Se il codice operativo IJVM è DUP, POP oppure SWAP occorre modificare lo stack. L'istruzione DUP replica semplicemente la parola che si trova in cima allo stack. Dato che il valore di questa parola è già contenuto in TOS, l'operazione non fa altro che incrementare SP in modo che punti alla nuova locazione e memorizzare TOS in quella posizione. L'istruzione POP è altrettanto semplice e consiste nel decrementare SP per scartare la parola in cima allo stack. Tuttavia, per mantenere corretto il valore memorizzato in TOS, occorre leggere dalla memoria la nuova parola che si trova in cima allo stack per scriverla all'interno di TOS. Infine l'istruzione SWAP effettua lo scambio dei valori contenuti nelle due locazioni che si trovano in cima allo stack. L'operazione è facilitata dal fatto che TOS contiene già uno di questi valori, e non è necessario leggerlo dalla memoria. In seguito tratteremo in modo più dettagliato questa istruzione.

L'istruzione BIPUSH è un po' più complicata poiché, come mostra la Figura 4.18, il codice operativo è seguito da un solo byte, che deve essere interpretato come un intero con segno. Questo byte, che durante Main1 è già stato portato all'interno di MBR, deve essere esteso con segno fino a 32 bit e copiato all'interno di MDR. Infine SP viene incrementato e copiato in MAR, in modo da permettere la scrittura dell'operando in cima allo stack. Questo operando deve anche essere copiato nel registro TOS. Inoltre è importante notare che, prima di ritornare al programma principale, PC deve essere nuovamente incrementato in modo da rendere disponibile in Main1 il successivo codice operativo.

Consideriamo ora l'istruzione ILOAD. Come mostra la Figura 4.19(a) anche in questa istruzione il codice operativo è seguito da un byte. In questo caso si tratta però di un indice

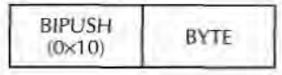


Figura 4.18 Formato dell'istruzione BIPUSH.

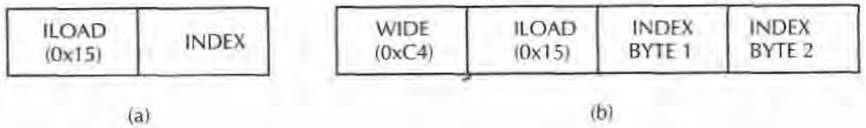


Figura 4.19 (a) `ILOAD` con un indice a 1 byte. (b) `WIDE ILOAD` con un indice a 2 byte.

(senza segno) usato per identificare nello spazio delle variabili locali la parola da mettere nello stack. Dato che si usa un solo byte è possibile distinguere soltanto 2^8 =256 parole, per la precisione le prime 256 parole dello spazio delle variabili locali. L'istruzione `ILOAD` richiede sia una lettura (per ottenere la parola) sia una scrittura (per porla in cima allo stack). Tuttavia per poter determinare l'indirizzo di lettura occorre sommare lo spiazzamento, contenuto in `MBR`, al valore di `LV`. Dato che è possibile accedere sia a `MBR` sia a `LV` solamente attraverso il bus B, inizialmente si copia `LV` in `H` (`iload1`), e poi si somma `MBR`. Dopo che il risultato è stato copiato in `MAR` può aver inizio una lettura (in `iload2`).

Tuttavia l'utilizzo di `MBR` come indice è leggermente diverso dall'uso che ne è stato fatto in `BIPUSH`. Nel caso di un indice il suo valore è sempre positivo e quindi il byte che lo rappresenta deve essere considerato come un intero senza segno; nel caso dell'istruzione `BIPUSH` esso è stato invece interpretato come un intero a 8 bit con segno. L'interfaccia tra `MBR` e il bus è stata attentamente progettata in modo da consentire entrambi i tipi di operazione. Nel caso di `BIPUSH` (intero con segno a 8 bit) l'operazione corretta è l'estensione con segno, in cui il bit più a sinistra del primo byte di `MBR` viene copiato nei 24 bit più alti del bus B. Nel caso di `ILOAD` (intero senza segno a 8 bit) l'operazione corretta è invece un riempimento con valori 0; in questo caso tutti e 24 i bit più alti del bus B vengono impostati a 0. Usando segnali separati è possibile distinguere le due operazioni e indicare quale deve essere effettuata (vedi Figura 4.6). Nel microcodice ciò è indicato da `MBR` (estensione con segno, come in `BIPUSH 3`) oppure da `MBRU` (senza segno, come in `iload2`).

Mentre si attende che la memoria fornisca l'operando (in `iload3`), `SP` viene incrementato in modo che il suo valore indichi la locazione in cui memorizzare il risultato, ovvero la nuova cima dello stack. Questo valore viene copiato anche in `MAR` per preparare la scrittura dell'operando nello stack. `PC` deve essere nuovamente incrementato per prelevare il codice operativo successivo (in `iload4`). Infine `MDR` viene copiato in `TOS` in modo da far correttamente riferimento alla nuova cima dello stack (in `iload5`).

`ISTORE` è l'operazione inversa di `ILOAD` e consiste nel rimuovere una parola dalla cima dello stack e nel memorizzarla nella locazione specificata dalla somma tra `LV` e l'indice contenuto nell'istruzione. `ISTORE` ha lo stesso formato di `ILOAD`, mostrato nella Figura 4.19(a), tranne per il fatto che il codice operativo è 0x36 invece che 0x15. Dato che si conosce già (in `TOS`) il valore della parola in cima allo stack, si potrebbe pensare che sia sufficiente memorizzare il contenuto di `TOS` direttamente in memoria; in realtà il comportamento dell'istruzione è leggermente più complesso. Occorre infatti prelevare la nuova parola che si trova in cima allo stack e per far ciò è necessario effettuare una lettura oltre alla scrittura; tuttavia non è importante l'ordine secondo il quale vengono eseguite queste due operazioni di memoria (se possibile possono anche essere eseguite in parallelo).

Sia `ILOAD` sia `ISTORE` possono accedere solamente alle prime 256 variabili locali. Ovviamente, anche se nella maggior parte dei programmi questo limite potrebbe essere superiore allo spazio effettivamente necessario, è fondamentale avere la possibilità di accedere a una variabile ovunque essa si trovi (nello spazio delle variabili locali). Per superare questa restrizione IJVM utilizza lo stesso meccanismo impiegato in JVM: si usa uno speciale codice operativo, `WIDE`, conosciuto come **byte di prefisso**, seguito dal codice operativo di `ILOAD` oppure da quello di `ISTORE`. Quando si incontra questa sequenza si modificano le definizioni di `ILOAD` e `ISTORE`, in modo che l'indice che segue il codice operativo sia un valore a 16 bit invece che a 8 bit. L'uso del codice operativo `WIDE` è mostrato nella Figura 4.19(b).

`WIDE` viene decodificato nel solito modo, effettuando una diramazione che porta all'istruzione `wide1` in cui inizia il microcodice specifico per la gestione di questo codice operativo. Anche se il codice operativo sul quale si vuole effettuare l'"allargamento" è già presente in `MBR`, `wide1` preleva il primo byte che lo segue, dato che la logica del microprogramma si aspetta sempre di trovarlo in quella posizione. In `wide2` viene effettuata una seconda diramazione in base al byte che segue `WIDE`. Tuttavia, dato che sia `WIDE ILOAD` sia `WIDE ISTORE` richiedono microcodice diverso rispetto a `ILOAD` e `ISTORE`, la seconda diramazione non può utilizzare il codice operativo come indirizzo di destinazione, come invece fa `Main1`.

In `wide2` viene invece calcolato l'OR tra il codice operativo e 0x100, e il risultato viene memorizzato in `MPC`. Come conseguenza l'interpretazione di `WIDE ILOAD` comincia in 0x115 (al posto di 0x15) e l'interpretazione di `WIDE ISTORE` comincia in 0x136 (al posto di 0x36), e così via. Così facendo ogni codice operativo di tipo `WIDE` inizia a un indirizzo che è 256 (cioè, 0x100) parole più in alto rispetto al corrispondente codice operativo di tipo normale. La Figura 4.20 mostra la sequenza iniziale di microistruzioni sia per `ILOAD` sia per `WIDE ILOAD`.

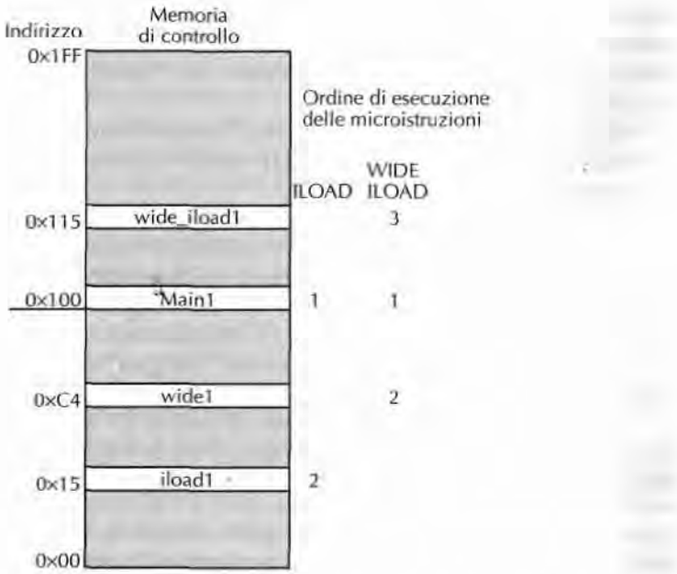


Figura 4.20 La sequenza iniziale di microistruzioni per `ILOAD` e `WIDE ILOAD`. Gli indirizzi sono puramente esemplificativi.

Raggiunto il codice che implementa WIDE ILOAD (0x115), esso differisce dal normale codice di ILOAD soltanto per il fatto che l'indice deve essere costruito concatenando 2 byte invece che estendendo con segno un unico byte. La concatenazione e la successiva addizione devono essere compiute in passi distinti, il primo dei quali consiste nel copiare in H il byte 1 dell'indice traslato a sinistra di 8 bit. Dato che l'indice è un intero senza segno, MBR viene poi esteso con valori zero utilizzando MBRU. Successivamente viene sommato il secondo byte (l'operazione di addizione è identica alla concatenazione, dato che ora il byte meno significativo di H vale zero, garantendo quindi che non ci sarà alcun riporto tra i byte) e il risultato viene memorizzato nuovamente in H. A partire da questo punto l'operazione può procedere, esattamente come se fosse il caso normale di ILOAD. Piuttosto che duplicare le ultime istruzioni di ILOAD (da iload3 a iload5), abbiamo semplicemente introdotto un salto da wide_iload4 a iload3. Si noti tuttavia che, durante l'esecuzione dell'istruzione, PC deve essere incrementato due volte affinché punti correttamente al successivo codice operativo. Sia l'istruzione ILOAD, sia la sequenza WIDE_ILOAD lo incrementano una volta.

La stessa situazione si verifica nel caso di WIDE_ISTORE: dopo aver eseguito le prime quattro microistruzioni (da wide_istore1 a wide_istore4), la sequenza è identica a quella di ISTORE a partire dalla terza istruzione. Viene quindi introdotto un salto da wide_istore4 a istore3.

La successiva istruzione d'esempio che prendiamo in considerazione è LDC_W. Questo codice operativo differisce da ILOAD per due aspetti. In primo luogo, l'istruzione ha uno spiazzamento di 16 bit senza segno (come la versione "allargata" di ILOAD). In secondo luogo, l'indice è relativo a CPP e non a LV, dato che la sua funzione consiste nel leggere dalla porzione costante di memoria e non dal blocco delle variabili locali. (In realtà esiste anche una versione corta di LDC_W, chiamata LDC; non l'abbiamo però inclusa in IJVM, dato che la versione lunga comprende già al suo interno tutti i possibili casi di quella corta, anche se richiede 3 byte al posto di soli 2.)

L'istruzione IINC è, oltre a ISTORE, l'unica istruzione IJVM che modifica una variabile locale. Come mostra la Figura 4.21, essa lo fa includendo due operandi, ciascuno dei quali è lungo 1 byte.

L'istruzione IINC utilizza INDEX per specificare lo spiazzamento rispetto all'inizio del blocco delle variabili locali, legge la variabile, la incrementa in base a un valore, CONST, contenuto nell'istruzione e memorizza il risultato nella stessa locazione. Si osservi che questa istruzione può incrementare di un valore negativo, nel senso che CONST è una costante a 8 bit con segno, il cui valore è dunque compreso tra –128 e +127. La versione completa di JVM comprende una versione "allargata" di IINC in cui ciascun operando è lungo 2 byte.

Arriviamo ora alla prima istruzione di salto di IJVM: GOTO. L'unica funzione di questa istruzione è quella di modificare il valore di PC, in modo che la successiva istruzione IJVM da eseguire sia quella che si trova nell'indirizzo calcolato sommando lo spiazzamento a 16 bit

Figura 4.21 L'istruzione INC ha due diversi campi per gli operandi.

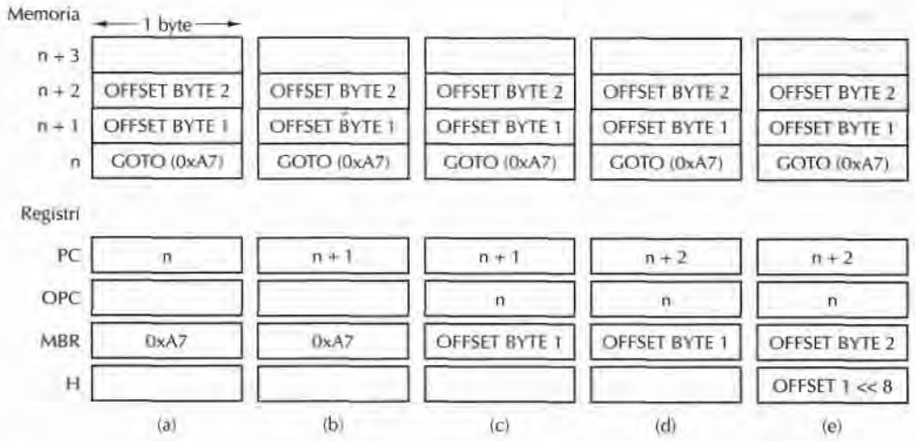


Figura 4.22 Situazione all'inizio di alcune microistruzioni. (a) Main1. (b) goto1. (c) goto2. (d) goto3. (e) goto4.

(con segno) all'indirizzo del codice operativo del salto.Una complicazione è data dal fatto che lo spiazzamento è relativo al valore che PC aveva all'inizio della decodifica dell'istruzione e non a quello prelevato dopo i due byte dello spiazzamento.

Per chiarire questo punto osserviamo nella Figura 4.22(a) la situazione all'inizio di Main1. Il codice operativo è già presente in MBR, ma PC non è ancora stato incrementato. Nella Figura 4.22(b) vediamo la situazione all'inizio di goto1. A questo punto PC è stato incrementato, ma il primo byte dello spiazzamento non è ancora stato prelevato e salvato in MBR. Una microistruzione dopo ci troviamo nel caso della Figura 4.22(c) in cui il vecchio PC, che punta al codice operativo, è stato salvato in OPC e il primo byte dello spiazzamento è stato memorizzato in MBR. Il valore salvato in OPC è necessario, in quanto lo spiazzamento dell'istruzione IJVM GOTO è relativo a esso e non al valore corrente di PC. Questo è in realtà il motivo principale per il quale abbiamo bisogno del registro OPC.

La microistruzione in goto2 inizia a prelevare il secondo byte dello spiazzamento, portando, all'inizio di goto3, alla situazione mostrata nella Figura 4.22(d). Dopo che il primo byte dello spiazzamento è stato traslato a sinistra di 8 bit e copiato in H, arriviamo a goto4 e allo stato della Figura 4.22(e). Ora il primo byte dello spiazzamento si trova in H traslato a sinistra di 8 bit, il secondo byte dello spiazzamento è in MBR e l'indirizzo base è salvato in OPC. In goto5 otteniamo il nuovo indirizzo da memorizzare in PC costruendo in H lo spiazzamento completo a 16 bit e sommandolo alla base. Si faccia attenzione al fatto che in goto4 utilizziamo MBRU al posto di MBR, dato che non vogliamo estendere con segno il secondo byte. Inoltre lo spiazzamento a 16 bit viene costruito calcolando in realtà l'OR logico tra le due metà. Infine, prima di tornare all'istruzione Main1, dobbiamo prelevare il successivo codice operativo, dato che all'inizio di ogni iterazione del ciclo principale il microprogramma si aspetta di trovarlo all'interno di MBR.

Nell'istruzione IJVM goto gli spiazzamenti sono valori a 16 bit con segno, compresi tra un minimo di –32768 e un massimo di +32767. Questo significa che non è pos-

sibile effettuare salti verso etichette la cui distanza superi questi limiti. Questa proprietà può essere vista come un bug oppure come una delle caratteristiche funzionali di IJVM (e anche di JVM). Chi sostiene che sia un bug ritiene che la definizione di JVM non dovrebbe restringere lo stile di programmazione. Chi invece sostiene che sia una caratteristica di JVM crede che il lavoro di molti programmatori migliorerebbe radicalmente se fossero inseguiti dall'incubo di veder apparire questo temibile messaggio da parte del compilatore

```
Il programma è troppo grande e complesso. Occorre riscriverlo.
La compilazione è annullata.
```

Sfortunatamente (dal nostro punto di vista) questo messaggio appare solamente quando una clausola `then` o `else` supera 32 KB, corrispondenti a circa 50 pagine di codice Java.

Consideriamo ora le tre istruzioni di salto condizionale di IJVM: `IFLT`, `IFEQ` e `IF_ICMPEQ`. Le prime due estraggono la parola in cima allo stack ed effettuano una diramazione se e solo se è rispettivamente minore di zero oppure uguale a zero. L'istruzione `IF_ICMPEQ` estrae invece le due parole in cima allo stack e compie una diramazione se e solo se sono uguali. In tutti e tre i casi è necessario leggere la nuova parola che si trova in cima allo stack e memorizzarla in `TOS`.

Per tutte e tre queste istruzioni il controllo è simile. Inizialmente l'operando, o gli operandi, vengono salvati nei registri; successivamente viene letto il nuovo valore in cima allo stack e inserito in `TOS`; infine vengono effettuati il test e la diramazione. Come prima istruzione consideriamo `IFLT`. La parola da testare è già presente in `TOS`, ma dato che `IFLT` estrae una parola dallo stack occorre leggere la nuova parola che si trova in cima allo stack per memorizzarla in `TOS`. Questa lettura inizia in `iflt1`. Durante `iflt2` viene temporaneamente salvata in `OPC` la parola da testare, in modo che il nuovo valore possa essere memorizzato in `TOS` senza perdere il valore corrente. Infine, durante `iflt4`, la parola che si vuole testare, salvata in `OPC`, viene fatta passare nella ALU senza memorizzarla in modo da testare il bit `N`. Questa microistruzione contiene anche una diramazione che sceglie la destinazione `T` se il test ha successo, mentre la `F` in caso contrario.

Se il test ha successo il resto dell'operazione è essenzialmente uguale all'istruzione `GOTO`; la sequenza continua quindi con `goto2`, in un punto intermedio della sequenza di `GOTO`. Se invece il test fallisce occorre eseguire una sequenza di breve durata (`F`, `F2` e `F3`) per scavalcare il resto dell'istruzione (lo spiazzamento) prima di ritornare a `Main1` e continuare con l'istruzione successiva.

Il codice in `ifeq2` e `ifeq3` segue la stessa logica, con l'unica differenza che si usa il bit `Z` invece del bit `N`. In entrambi i casi è compito dell'assemblatore di `MAL` riconoscere che gli indirizzi `T` e `F` sono speciali e assicurarsi che siano posizionati nella memoria di controllo in modo che i loro indirizzi differiscano soltanto nel bit più a sinistra.

Anche la logica dell'istruzione `IF_ICMPEQ` è più o meno simile a quella di `IFEQ`, tranne per il fatto che in questo caso dobbiamo leggere anche il secondo operando. Il secondo operando viene caricato in `H` durante `if_icmpeq3`, quando inizia la lettura della nuova parola in cima allo stack. Anche questa volta la parola che si trova in cima allo stack viene salvata in `OPC` e quella nuova viene memorizzata in `TOS`. Infine il test `if_icmpeq6` è simile a quello in `ifeq4`.

Consideriamo ora l'implementazione di `INVOKEVIRTUAL` e `IRETURN`; com'è stato descritto nel Paragrafo 4.2.3 sono le istruzioni che permettono di richiamare una procedura e di restituire il controllo al chiamante. `INVOKEVIRTUAL` è l'istruzione più complessa implementata in IJVM ed è costituita da una sequenza di 22 microistruzioni, mostrate nella Figura 4.12. L'istruzione utilizza uno spiazzamento a 16 bit per determinare l'indirizzo del metodo da invocare. Nella nostra implementazione lo spiazzamento è semplicemente un valore di distanza all'interno della porzione costante di memoria; questa locazione punta al metodo da invocare. Non dimentichiamo però che i primi 4 byte di ciascun metodo *non* sono istruzioni; essi contengono infatti due puntatori a 16 bit. Il primo fornisce il numero di parole usate per i parametri (compreso `OBJREF`, vedi Figura 4.12). Il secondo fornisce invece la dimensione dell'area delle variabili locali espressa come numero di parole. Questi campi sono prelevati attraverso la porta a 8 bit e vengono assemblati come se fossero due spiazzamenti a 16 bit presenti all'interno di un'istruzione.

Successivamente vengono memorizzate delle informazioni di collegamento necessarie per poter riportare la macchina allo stato precedente. Queste informazioni comprendono l'indirizzo d'inizio di ciascuna area delle variabili locali oltre al precedente valore di `PC`; esse sono memorizzate immediatamente sopra l'area delle variabili locali appena creata e sotto il nuovo stack. Infine, prima di ritornare a `Main1` e poter iniziare l'esecuzione della nuova istruzione, viene prelevato il codice operativo dell'istruzione successiva e `PC` viene incrementato.

`IRETURN` è un'istruzione semplice che non contiene alcun operando. Essa utilizza semplicemente l'indirizzo memorizzato nella prima parola dell'area delle variabili locali per recuperare le informazioni di collegamento. In seguito reimposta `SP`, `LV` e `PC` ai loro precedenti valori e, come mostra la Figura 4.13, copia in cima allo stack originale il valore di ritorno presente in cima allo stack corrente.

4.4 Progettazione del livello di microarchitettura

La progettazione del livello di microarchitettura, come tutto ciò che riguarda i calcolatori, è caratterizzata da compromessi e bilanciamenti. Sono molte le caratteristiche di un calcolatore che si vorrebbero ottimizzare: velocità, costi, affidabilità, facilità di utilizzo, consumo energetico e dimensioni fisiche. Fra tutti i possibili compromessi ce n'è uno in particolare che determina le principali scelte che un progettista deve compiere: l'equilibrio tra la velocità della macchina e i suoi costi. In questo paragrafo analizzeremo il problema nel dettaglio per vedere quali elementi occorre bilanciare, come riuscire a ottenere prestazioni elevate e a quale costo, in termini di hardware e complessità, ciò sia possibile.

4.4.1 Velocità/costi

Ultimamente l'impiego di nuove tecnologie ha prodotto nei calcolatori il maggior incremento di velocità finora registrato; tuttavia questo aspetto esula dagli scopi del libro, dato che siamo maggiormente interessati agli aumenti di velocità dovuti all'architettura. I miglioramenti dovuti a modifiche dell'architettura dei calcolatori, pur non essendo stati stupefacenti quanto quelli dovuti all'impiego di circuiti più rapidi, hanno avuto comunque un impatto notevole. La velocità può essere misurata in più modi; in ogni caso, una volta scelto un ISA e stabilita la tecnologia dei circuiti, esistono principalmente tre approcci tramite i quali è possibile aumentare la velocità di esecuzione.

1. Ridurre il numero di cicli di clock necessari per eseguire un'istruzione.
2. Semplificare l'organizzazione in modo che il ciclo di clock possa essere più breve.
3. Sovrapporre l'esecuzione delle istruzioni.

Mentre i primi due punti sono ovvi, esiste un numero sorprendentemente alto di possibili progettazioni che possono influire in modo drastico sul numero di cicli di clock, sul periodo di clock o, molto spesso, su entrambi i fattori. In questo paragrafo daremo un esempio di come la codifica e la decodifica di un'istruzione siano in grado di modificare il ciclo di clock.

Il numero di cicli di clock necessario per eseguire un insieme di operazioni è conosciuto con il nome di **lunghezza del percorso**. In alcuni casi la lunghezza del percorso può essere accorciata aggiungendo dei componenti hardware specializzati. Se per esempio si aggiunge al PC un incrementatore (ovvero un sommatore in cui un addendo è sempre impostato a 1) è possibile eliminare alcuni cicli, dato che non è più necessario utilizzare la ALU per far avanzare il suo valore; in questo caso il prezzo da pagare è rappresentato dall'hardware aggiuntivo. Tuttavia questa possibilità non aiuta tanto quanto ci si potrebbe aspettare. Per la maggior parte delle istruzioni, durante gli stessi cicli spesi per incrementare PC, viene infatti svolta un'operazione di lettura; di conseguenza l'esecuzione dell'istruzione successiva non può essere anticipata, dato che dipende dal dato che deve giungere dalla memoria.

Se si intende ridurre il numero di cicli necessari per prelevare le istruzioni occorre qualcosa di più rispetto al solo circuito che incrementa il PC. La tecnica più efficace per una significativa accelerazione del prelievo delle istruzioni è la terza, ovvero la sovrapposizione dell'esecuzione di più istruzioni. Separare i componenti del circuito relativi al prelievo dell'istruzione (la porta a 8 bit della memoria e i registri MBR e PC) è più efficace se si rende quest'unità indipendente, dal punto di vista funzionale, dal percorso dati. In questo modo l'unità può prelevare autonomamente il successivo codice operativo o operando, anche in modo asincrono rispetto al resto della CPU. In tal modo è possibile prelevare in anticipo una o più istruzioni.

Nell'esecuzione di molte istruzioni una delle operazioni più costose in termini di tempo consiste nel prelevare uno spiazzamento a 2 byte, nell'estenderlo in modo appropriato e nel depositarlo nel registro H in preparazione di un'addizione (ciò serve, per esempio, per calcolare l'indirizzo PC ± *n* a cui porta una diramazione). Una possibile soluzione potrebbe essere quella di rendere la porta della memoria larga 16 bit. Purtroppo però questa scelta complicherebbe l'operazione; la larghezza della memoria è infatti di 32 bit e, dato che i 16 bit richiesti potrebbero superare i limiti della parola, anche una singola lettura di 32 bit non riuscirebbe necessariamente a prelevare entrambi i byte richiesti.

La sovrapposizione dell'esecuzione delle istruzioni è di gran lunga la soluzione più interessante e offre le maggiori opportunità per un incremento drastico della velocità. La semplice sovrapposizione del prelievo e dell'esecuzione di un'istruzione ha un'efficacia sorprendente. Esistono anche alcune tecniche più sofisticate che si spingono oltre, permettendo per esempio di sovrapporre l'esecuzione di più istruzioni. Questa idea è infatti il cuore delle architetture dei calcolatori moderni. In seguito illustreremo alcune delle tecniche più semplici per sovrapporre l'esecuzione delle istruzioni e spiegheremo quali sono le motivazioni che portano all'adozione di tecniche ancora più sofisticate.

La velocità costituisce soltanto metà del problema; l'altra è rappresentata dai costi. Anche il costo può essere misurato in vari modi, ma è difficile darne una precisa definizione. Alcune misure si basano semplicemente sul numero di componenti (il che aveva un senso quando i processori venivano costruiti a partire da componenti acquistati separatamente e poi assemblati). Oggi l'intero processore è invece realizzato su un unico chip. Resta comunque vero che chip più complessi sono più costosi rispetto a quelli più semplici e piccoli. Anche se è possibile contare i singoli componenti, come i transistor, le porte logiche e le unità funzionali, spesso questo valore non è importante quanto invece lo è l'area occupata sul circuito integrato. Maggiori sono le funzioni incluse nel processore e più grande diventa il chip; all'aumentare delle sue dimensioni i costi di produzione crescono, ma in modo molto più veloce rispetto alla sua area. Per questo motivo spesso i progettisti parlano di costi in termini di "proprietà terriera", riferendosi in questo modo all'area richiesta dal circuito (presumibilmente misurata in pico-acri).

Uno dei circuiti che nella storia è stato studiato in modo più accurato è il sommatore. Sono stati realizzati migliaia di progetti e quelli più veloci, oltre a essere molto complessi, hanno prestazioni decisamente migliori rispetto a quelli più lenti. Il progettista di sistema deve decidere se l'aumento di velocità giustifica l'aumento della "proprietà terriera".

I sommatori non sono però gli unici componenti per i quali esistono più scelte possibili. Praticamente qualsiasi componente del sistema può essere progettato per funzionare più velocemente o più lentamente, con un'influenza sul costo. La sfida che si presenta al progettista è quella di identificare i componenti che possono migliorare il sistema nel modo più marcato, aumentandone la velocità. È interessante sapere che spesso un singolo componente può essere sostituito da uno molto più veloce producendo però soltanto un effetto limitato, se non addirittura nullo, sulla velocità globale del sistema. Nei paragrafi successivi analizzeremo alcuni problemi di progettazione e i relativi compromessi nella scelta dei componenti.

Per determinare a quale velocità può arrivare il clock, uno dei fattori chiave è la quantità di lavoro da eseguire durante ogni ciclo. Ovviamente all'aumentare del lavoro da eseguire il ciclo di clock si allunga. In realtà però la cosa non è così semplice, dato che l'hardware è sufficientemente evoluto per eseguire più compiti in parallelo; ciò che determina effettivamente la lunghezza del ciclo di clock è quindi la sequenza di operazioni che devono essere eseguite *serialmente* durante ogni singolo ciclo.

Un aspetto che può essere controllato è la quantità di operazioni di decodifica da eseguire. Per esempio, nella Figura 4.6 abbiamo visto che, nonostante qualunque dei nove registri potesse essere caricato nella ALU dal bus B, all'interno della parola della microistruzione avevamo bisogno solamente di 4 bit per indicare il registro selezionato. Purtroppo questo risparmio ha i suoi contro: il ritardo introdotto dal circuito di decodifica nel percorso dati. Ciò significa che il registro selezionato per portare i propri dati sul bus B riceverà il comando leggermente in ritardo e solo in seguito i suoi dati verranno effettivamente messi sul bus. Ciò provoca un effetto a catena: la ALU riceverà i propri input, e quindi produrrà il risultato, leggermente in ritardo e, con lo stesso ritardo, il risultato verrà scritto sul bus C. Dato che spesso questo ritardo determina quanto deve essere lungo il ciclo di clock, ciò potrebbe limitare la sua frequenza, costringendo quindi l'intero calcolatore a funzionare a una velocità leggermente inferiore. C'è dunque un bilanciamento tra velocità e costi. La ridu-

zione della memoria di controllo di 5 bit per parola avviene a spese di un rallentamento del clock. Il progettista, quando deve compiere la scelta più appropriata, deve tener ben presente gli obiettivi della progettazione. Se si vuole implementare un sistema ad alte prestazioni, l'utilizzo di un decodificatore non è probabilmente una buona idea, ma lo è per i sistemi a basso costo.

4.4.2 Riduzione della lunghezza del percorso di esecuzione

La macchina Mic-1 è stata progettata per essere allo stesso tempo moderatamente semplice e moderatamente veloce, anche se questi obiettivi sono chiaramente in conflitto l'uno con l'altro. In poche parole, le macchine semplici non sono veloci e le macchine veloci non sono semplici. La CPU di Mic-1 utilizza inoltre una quantità minima di hardware: 10 registri, la semplice ALU della Figura 3.19 replicata 32 volte, uno shifter, un decodificatore, una memoria di controllo e alcuni bit qua e là, necessari per tenere insieme i diversi componenti. L'intero sistema potrebbe essere costruito con meno di 5000 transistor, la ROM di controllo e la memoria centrale.

Dopo aver visto com'è possibile implementare IJVM mediante microcodice e con poco hardware è giunto ora il momento di analizzare delle alternative più veloci. Studieremo ora alcuni modi per ridurre il numero di microistruzioni delle istruzioni ISA (cioè, per ridurre la lunghezza del percorso di esecuzione).

Unione del ciclo d'interpretazione con il microcodice

In Mic-1 il ciclo principale consiste in una microistruzione da eseguire all'inizio di ogni istruzione IJVM. In alcuni casi è possibile sovrapporla all'istruzione precedente e in realtà in parte questo è già stato fatto. Notiamo infatti che quando `Main1` viene eseguito il codice operativo da interpretare è già presente all'interno di `MBR`. Il codice operativo si trova già nel registro, in quanto è già stato prelevato o dal precedente ciclo principale oppure durante l'esecuzione dell'istruzione precedente.

È possibile spingere ancora più in avanti quest'idea, che consiste nel sovrapporre l'inizio dell'istruzione. In alcuni casi è possibile addirittura ridurre il ciclo principale fino a farlo scomparire. Ciò può verificarsi nel modo seguente. Consideriamo le sequenze di microistruzioni che terminano con un salto a `Main1`. In ognuno di questi punti l'istruzione del ciclo principale può essere spostata alla fine della sequenza (piuttosto che all'inizio della sequenza successiva), replicando così in più punti del microcodice la diramazione a più destinazioni (mantenendo però sempre lo stesso insieme di destinazioni). In alcuni casi la microistruzione `Main1` può essere unita alle microistruzioni precedenti, dato che non sempre queste vengono utilizzate completamente.

Etichetta	Operazioni	Commenti
pop1	MAR = SP = SP – 1; rd	Legge la parola sotto la cima dello stack
pop2		Attende che il nuovo TOS sia letto dalla memoria
pop3	TOS = MDR; goto Main1	Copia la nuova parola in TOS
Main1	PC = PC + 1; fetch; goto (MBR)	MBR contiene il codice operativo; prelievo del byte successivo; diramazione

Figura 4.23 Sequenza della versione originale del microprogramma per l'esecuzione di POP.

Etichetta	Operazioni	Commenti
pop1	MAR = SP = SP – 1; rd	Legge la parola sotto la cima dello stack
Main1.pop	PC = PC + 1; fetch	MBR contiene il codice operativo; prelievo del byte successivo
pop3	TOS = MDR; goto (MBR)	Copia la nuova parola in TOS; diramazione a seconda del codice operativo

Figura 4.24 Sequenza della versione migliorata del microprogramma per l'esecuzione di POP.

La Figura 4.23 mostra la sequenza dinamica di microistruzioni nel caso di un'istruzione `POP`. Il ciclo principale si verifica prima e dopo ciascuna istruzione, anche se la figura mostra solamente l'occorrenza successiva all'istruzione `POP`. Notiamo che l'esecuzione di questa istruzione richiede quattro cicli di clock: tre per le microistruzioni specifiche di `POP` e uno per il ciclo principale.

Nella Figura 4.24 la sequenza è stata ridotta a tre istruzioni, unendo le istruzioni del ciclo principale alle altre; per eseguire alcune delle operazioni di `Main1` è stato infatti sfruttato il ciclo di clock `pop2` in cui la ALU non veniva utilizzata. È importante notare che questa sequenza termina con una diramazione che porta l'esecuzione direttamente nel punto del codice specifico per l'istruzione successiva: complessivamente tre cicli sono sufficienti. Questa piccola modifica riduce di un ciclo il tempo di esecuzione della successiva microistruzione. Ciò equivale ad accelerare il clock da 250 MHz (microistruzioni da 4 ns) a 333 MHz (microistruzioni da 3 ns), senza realmente modificarne la temporizzazione.

L'istruzione `POP` si presta particolarmente bene a questo trattamento, dato che nel mezzo della sua esecuzione è presente un ciclo inattivo in cui la ALU non viene utilizzata. Dato che il ciclo principale richiede l'uso della ALU, se si vuole ridurre la lunghezza di un'istruzione, è necessario trovare al suo interno un ciclo in cui la ALU è inattiva. Questi cicli "morti" non sono comuni, ma quando si verificano vale la pena unire `Main1` alla fine di ciascuna sequenza di microistruzioni. Il prezzo da pagare è solamente un piccolo incremento della memoria di controllo. È possibile formulare in questo modo la prima tecnica che ci permette di ridurre la lunghezza del percorso:

unire il ciclo d'interpretazione alla fine delle sequenze di microcodice.

Architettura a tre bus

Cos'altro possiamo fare per ridurre la lunghezza del percorso di esecuzione? Un'altra semplice modifica consiste nell'utilizzo di due bus di input, A e B, per la ALU, per un totale di tre bus. Tutti (almeno la maggior parte) dei registri dovrebbero avere accesso a entrambi i bus di input. Il vantaggio di avere due bus di input è che in questo modo diventa possibile sommare fra loro, in unico ciclo, due registri qualsiasi. Per comprendere il valore di questa funzionalità consideriamo l'implementazione di `ILOAD` per Mic-1 (Figura 4.25).

Possiamo osservare che durante `iload1` `LV` è copiato all'interno di `H`. Ciò avviene per l'unico motivo di poterlo successivamente sommare a `MBRU` durante `iload2`. Nella nostra architettura originale a due bus non esiste alcun modo per sommare direttamente due registri arbitrari; l'unica soluzione consiste nel copiarne inizialmente uno in `H`. Nella nuova architettura a tre bus possiamo invece risparmiare un ciclo, come si può vedere nella Figura 4.26. In questo caso abbiamo sommato il ciclo d'interpretazione a `ILOAD`, ma così facendo non abbiamo né aumentato né diminuito la lunghezza del percorso di esecuzione.

Etichetta	Operazioni	Commenti
iload1	H = LV	MBR contiene l'indice; copia LV in H
iload2	MAR = MBRU + H; rd	MAR = indirizzo della variabile locale da inserire nello stack
iload3	MAR = SP = SP + 1	SP punta alla nuova cima dello stack; prepara la scrittura
iload4	PC = PC + 1; fetch; wr	Incrementa PC; prelievo del successivo codice operativo; scrive in cima allo stack
iload5	TOS = MDR; goto Main1	Aggiorna TOS
Main1	PC = PC + 1; fetch; goto (MBR)	MBR contiene il codice operativo; prelievo del byte successivo; diramazione

Figura 4.25 Codice di Mic-1 per l'esecuzione di ILOAD.

Tuttavia l'aggiunta del terzo bus ha ridotto il tempo totale di esecuzione di `ILOAD`, portandolo da sei a cinque cicli. A questo punto abbiamo a disposizione una seconda tecnica che ci permette di ridurre la lunghezza del percorso:

passare da un'architettura a due bus a una a tre bus.

Unità di prelievo dell'istruzione

Entrambe le tecniche precedenti sono valide, ma per ottenere un netto miglioramento delle prestazioni abbiamo bisogno di qualcosa di molto più radicale. Facciamo un passo indietro e analizziamo le parti comuni a tutte le istruzioni, ovvero il prelievo e la decodifica dei campi. Per ogni istruzione si possono verificare le seguenti operazioni:

1. il valore di PC viene fatto passare attraverso la ALU per incrementarlo
2. il valore di PC è utilizzato per prelevare il byte successivo nel flusso delle istruzioni
3. gli operandi sono letti dalla memoria
4. gli operandi sono scritti in memoria
5. la ALU esegue una computazione e i risultati vengono memorizzati.

Se un'istruzione contiene campi aggiuntivi (per gli operandi), questi vanno prelevati esplicitamente, un byte alla volta, e poi assemblati prima di poterli utilizzare. Prelevare e assemblare un operando impegna la ALU per almeno un ciclo per byte, per incrementare PC e in seguito assemblare l'indice, o lo spiazzamento, risultante. Durante praticamente ogni ciclo la ALU viene utilizzata, oltre che per il "lavoro" vero e proprio dell'istruzione, anche per una

Etichetta	Operazioni	Commenti
iload1	MAR = MBRU + LV; rd	MAR = indirizzo della variabile locale da inserire sullo stack
iload2	MAR = SP = SP + 1	SP punta alla nuova cima dello stack; prepara la scrittura
iload3	PC = PC + 1; fetch; wr	Incrementa PC; prelievo del successivo codice operativo; scrive in cima allo stack
iload4	TOS = MDR	Aggiorna TOS
iload5	PC = PC + 1; fetch; goto (MBR)	MBR contiene già il codice operativo; prelievo del byte dell'indice

Figura 4.26 Codice dell'architettura a tre bus per l'esecuzione di ILOAD.

grande varietà di operazioni relative al prelievo dell'istruzione e l'assemblaggio dei campi al suo interno.

Per poter sovrapporre il ciclo principale è necessario liberare la ALU da alcuni di questi compiti. Ciò potrebbe essere fatto introducendo una seconda ALU, anche se per molte operazioni non è realmente necessario disporre di tutte le funzionalità che fornisce una ALU completa. Possiamo notare infatti che in molti casi la ALU viene semplicemente usata come un cammino per copiare un valore da un registro a un altro. Questi cicli potrebbero essere eliminati introducendo percorsi dati aggiuntivi che non passino attraverso la ALU. Per esempio potrebbe essere vantaggioso creare un percorso da `TOS` a `MDR`, oppure da `MDR` a `TOS`, dato che la parola in cima allo stack viene scambiata frequentemente tra questi due registri.

In Mic-1 è possibile eliminare gran parte del carico di lavoro che grava sulla ALU creando un'unità indipendente che si occupa del prelievo e dell'elaborazione delle istruzioni. Questa unità, chiamata **IFU** (*Instruction Fetch Unit*, "unità di fetch delle istruzioni"), può incrementare il `PC` in modo indipendente e può prelevare i byte prima ancora che siano richiesti. Per implementare questa unità è sufficiente un incrementatore, cioè un circuito molto più semplice di un sommatore. Portando ancora più avanti l'idea di un'unità separata per la gestione delle istruzioni è possibile realizzare una IFU in grado di assemblare gli operandi a 8 e 16 bit, in modo che siano immediatamente pronti in qualsiasi momento vengano richiesti. Per poter far ciò esistono almeno due modi distinti.

1. La IFU può interpretare realmente ciascun codice operativo, determinando quanti campi aggiuntivi devono essere prelevati, e assemblarli in un registro pronto per essere utilizzato dall'unità di esecuzione principale.
2. La IFU può trarre vantaggio dal fatto che le istruzioni sono in realtà un flusso di byte e rendere disponibili, in ogni momento, le successive porzioni di 8 e 16 bit, anche se potrebbero non essere necessarie. Spetta poi all'unità di esecuzione principale richiedere i dati veramente necessari.

La Figura 4.27 mostra i principi fondamentali del secondo schema. Al posto di un solo registro `MBR` a 8 bit, ora ce ne sono due: `MBR1` a 8 bit e `MBR2` a 16 bit. La IFU tiene traccia del byte, o dei byte, utilizzati più recentemente dall'unità di esecuzione principale. Inoltre, allo stesso modo di quanto avveniva in Mic-1, rende disponibile all'interno di `MBR1` il byte successivo; la differenza è che in questo caso l'unità, non appena rileva che `MBR1` è stato letto, preleva il byte successivo e lo memorizza immediatamente all'interno di `MBR1`. Come in Mic-1 il registro ha due interfacce con il bus B: `MBR1` e `MBR1U`. La prima è estesa col segno a 32 bit, mentre la seconda è estesa con bit zero.

In modo analogo, `MBR2` fornisce le stesse funzionalità memorizzando i successivi 2 byte. Anche questo registro ha due interfacce con il bus B, `MBR2` e `MBR2U`, che forniscono rispettivamente l'estensione con il segno a 32 bit e l'estensione con bit zero.

La IFU è responsabile del prelievo di un flusso di byte. Essa compie il proprio lavoro utilizzando una convenzionale porta di memoria a 4 byte, prelevando in anticipo intere parole a 4 byte e caricando i byte consecutivi all'interno di un registro a scorrimento. Questo registro ha la funzione di mantenere una coda di byte provenienti dalla memoria per alimentare `MBR1` e `MBR2`, fornendo loro uno, o due byte, alla volta nell'ordine in cui sono stati prelevati.

`MBR1` contiene sempre il byte più vecchio del registro a scorrimento, mentre `MBR2` contiene i 2 byte più vecchi (quelli più a sinistra), che servono a formare un intero a 16 bit,

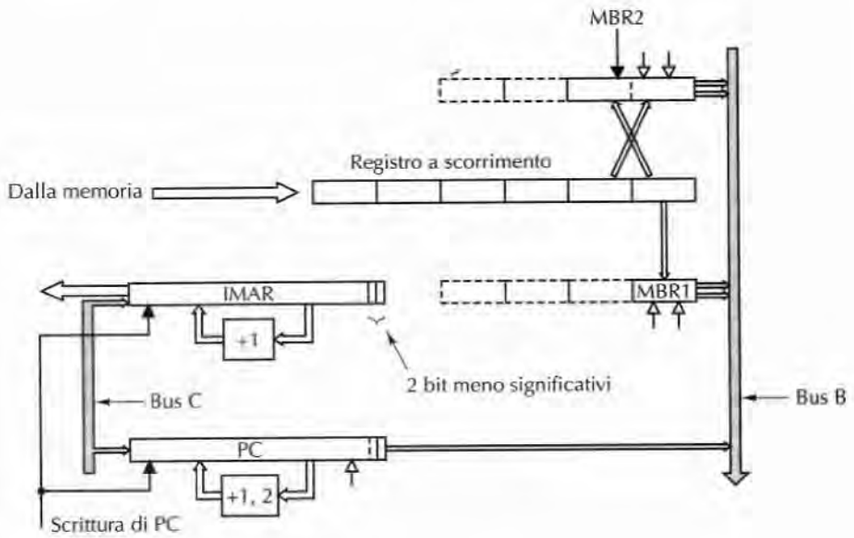


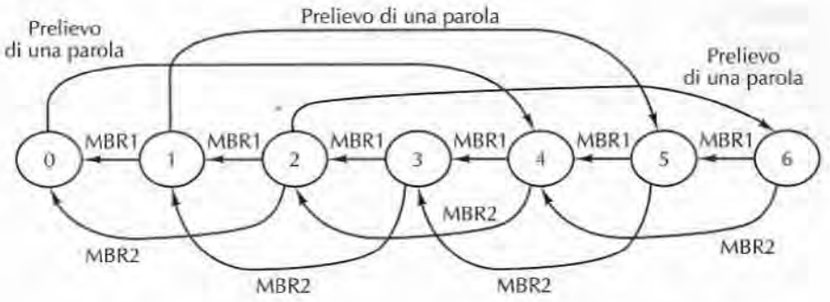
Figura 4.27 Unità di prelievo per Mic-1.

come mostrato nella Figura 4.19(b). I 2 byte di MBR2 possono anche provenire da parole diverse, dato che in memoria le istruzioni IJVM non sono allineate alle parole.

Ogni volta che viene letto MBR1, il registro a scorrimento viene traslato a destra di 1 byte, e di 2 quando viene letto MBR2. In seguito MBR1 e MBR2 vengono ricaricati rispettivamente con il byte o con la coppia di byte più vecchi. Se nella parte sinistra del registro a scorrimento c'è sufficiente spazio per contenere un'intera nuova parola, la IFU inizia un ciclo di memoria per leggerla. Assumiamo che, dopo la lettura di uno dei due registri MBR, questo venga nuovamente riempito all'inizio del ciclo successivo, in modo che sia possibile effettuare più letture durante cicli consecutivi.

Come mostra la Figura 4.28, è possibile modellare l'architettura della IFU mediante un **automa a stati finiti**. Questi automi sono costituiti da **stati** (rappresentati da cerchi) e da **transizioni** (rappresentate come archi che collegano uno stato con un altro). Ogni stato rappresenta una possibile situazione nella quale si può trovare l'automa. Il nostro automa ha sette stati, corrispondenti ai sette stati del registro a scorrimento della Figura 4.27. Gli stati sono identificati dagli interi da 0 a 6, che indicano il numero di byte presenti in un dato momento nel registro.

Un arco rappresenta un evento che si può verificare. Nel nostro caso ci sono tre possibili eventi. Il primo è la lettura di un byte da MBR1, e quando si verifica il registro a scorrimento viene attivato e trasla a destra per estrarre 1 byte, riducendo dunque lo stato di 1. Il secondo evento è una lettura di 2 byte da MBR2, che riduce lo stato di 2. Entrambe queste transizioni fanno sì che MBR1 e MBR2 vengano letti nuovamente. Quando l'automa si porta negli stati 0, 1 o 2, ha inizio un riferimento alla memoria per prelevare una nuova parola (assumendo che la memoria non sia già occupata nella lettura di una parola). Quando termina la lettura della parola l'automa avanza di 4 stati.



Transizioni
MBR1: avviene quando viene letta MBR1
MBR2: avviene quando viene letta MBR2
Prelievo di una parola: avviene quando viene letta una parola della memoria e vengono inseriti 4 byte nel registro a scorrimento

Figura 4.28 Automa a stati finiti che implementa la IFU.

Per un corretto funzionamento, l'automa deve bloccarsi quando gli viene fatta una richiesta che non è in grado di soddisfare, per esempio fornire il valore di MBR2 quando all'interno del registro c'è un solo byte e la memoria è ancora occupata nel prelievo della nuova parola. Inoltre l'automa può eseguire solamente un'azione alla volta e occorre quindi serializzare gli eventi in entrata. Infine, l'automa deve essere aggiornato ogni volta che PC viene modificato. Questi dettagli rendono l'automa più complicato di quanto abbiamo mostrato precedentemente; tuttavia molti dispositivi hardware sono progettati come automi a stati finiti.

La IFU possiede un proprio registro per l'indirizzo di memoria, chiamato IMAR, utilizzato per puntare all'indirizzo di memoria in cui si trova la parola da prelevare. Questo registro possiede un proprio incrementatore e non occorre quindi utilizzare la ALU per aggiornarlo (affinché si riferisca alla parola successiva). La IFU deve monitorare il bus C per rilevare quando il PC viene caricato e copiare di conseguenza il nuovo valore in IMAR. Dato che il nuovo valore del PC potrebbe non trovarsi sul limite di una parola, la IFU deve prelevare la parola corretta e regolare in modo appropriato il registro di scorrimento.

Utilizzando la IFU, il PC viene modificato dalla ALU solamente nel caso in cui sia necessario cambiare la natura sequenziale del flusso di byte delle istruzioni. Ciò si verifica quando viene soddisfatta la condizione di una diramazione e nel caso delle istruzioni INVOKEVIRTUAL e IRETURN.

È compito della IFU tenere aggiornato il PC, dato che il microprogramma non lo incrementa più in modo esplicito quando vengono prelevati i codici operativi. Per sapere quando occorre incrementare il PC, la IFU rileva quando viene consumato un byte del flusso dell'istruzione, cioè quando viene letto MBR1, MBR2, o le loro versioni senza segno. A seconda di quanti byte sono stati consumati un incrementatore aumenta il valore del PC di 1 oppure di 2. In questo modo il PC contiene sempre l'indirizzo del primo byte non ancora consumato. All'inizio di ogni istruzione, MBR contiene l'indirizzo del codice operativo dell'istruzione stessa.

Si noti la presenza di due incrementatori, con funzioni distinte. PC conta i *byte* e viene incrementato di 1 o di 2; IMAR conta invece le *parole* e viene incrementato soltanto di 1 (per 4 nuovi byte). Analogamente a MAR, anche IMAR è collegato all'indirizzo del bus in modo disallineato, con il bit 0 di IMAR connesso alla linea d'indirizzo 2, e così via, in modo da convertire implicitamente gli indirizzi di parole in indirizzi di byte.

Come vedremo tra poco in modo più dettagliato, il fatto di non dover incrementare PC nel ciclo principale rappresenta un grande vantaggio. Spesso infatti la microistruzione nella quale viene incrementato il PC compie poco lavoro aggiuntivo; se si eliminano queste istruzioni si riduce quindi il percorso di esecuzione. In questo caso il compromesso sta nell'aumento del numero di componenti hardware per disporre di una macchina più veloce. La nostra terza tecnica per ridurre la lunghezza del percorso prevede quindi

che le istruzioni siano prelevate dalla memoria da un'unità funzionale specializzata.

4.4.3 Architettura con *prefetching*: Mic-2

La IFU permette di ridurre in modo considerevole la lunghezza media del percorso di un'istruzione. In primo luogo elimina interamente il ciclo principale, dato che alla fine di ogni istruzione si effettua un semplice salto all'istruzione successiva. Inoltre risparmia l'utilizzo della ALU per l'incremento del PC, e riduce la lunghezza del percorso ogni volta che viene calcolato un indice o uno spiazzamento a 16 bit (dato che assembla il valore a 16 bit e lo fornisce direttamente alla ALU come un valore a 32 bit, evitando di doverlo assemblare all'interno di H). La Figura 4.29 mostra Mic-2, cioè una versione migliorata di Mic-1 in cui è stata aggiunta la IFU della Figura 4.27. La Figura 4.30 mostra il microcodice per la nuova macchina.

Per vedere un esempio del funzionamento di Mic-2 consideriamo IADD. La macchina preleva la seconda parola dello stack ed esegue l'addizione come prima, tranne per il fatto che, una volta terminata l'operazione, non è più necessario tornare a Main1 per incrementare PC e passare alla microistruzione successiva. Quando la IFU vede che durante iadd3 è stato effettuato un riferimento a MBR1, il suo registro a scorrimento trasla tutto il proprio contenuto a destra e ricarica sia MBR1 sia MBR2. L'unità effettua quindi una transizione verso uno stato che è inferiore di uno rispetto a quello corrente e, se il nuovo stato è 2, la IFU inizia a prelevare una nuova parola dalla memoria. Tutto ciò viene effettuato dall'hardware, senza alcun intervento del microcodice. Questo è il motivo che permette di ridurre IADD da quattro a tre microistruzioni.

Mic-2 permette di ottenere miglioramenti più marcati per alcune istruzioni piuttosto che altre. LDC_W passa da nove a tre microistruzioni; SWAP invece passa soltanto da otto a sei microistruzioni. Per migliorare le prestazioni globali del sistema ciò che conta veramente è il miglioramento che si può ottenere con le istruzioni usate più frequentemente. Le istruzioni più comuni sono ILOAD (da 6 a 3), IADD (da 4 a 3) e IF_ICMPEQ (da 13 a 10 quando la condizione è verificata; da 10 a 8 altrimenti). Anche se per misurare il miglioramento delle prestazioni bisognerebbe eseguire dei test, è tuttavia evidente che il guadagno sia significativo.

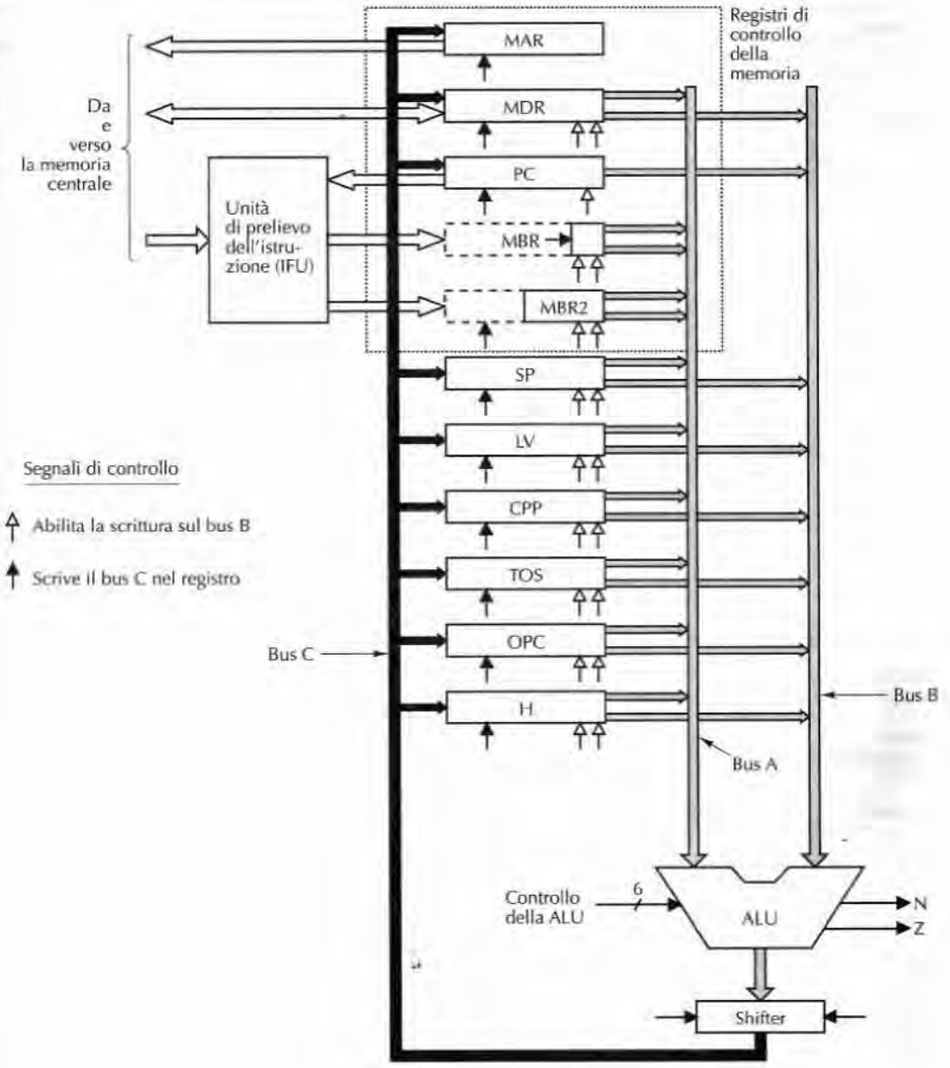


Figura 4.29 Percorso dati di Mic-2.

Etichetta	Operazioni	Commenti
nop1	goto (MBR)	Salto all'istruzione successiva
iadd1	MAR = SP = SP – 1; rd	Legge la parola sotto la cima dello stack
iadd2	H = TOS	H = cima dello stack
iadd3	MDR = TOS = MDR + H; wr; goto (MBR1)	Somma le due parole in cima allo stack; scrive in cima allo stack
isub1	MAR = SP = SP – 1; rd	Legge la parola sotto la cima dello stack
isub2	H = TOS	H = cima allo stack
isub3	MDR = TOS = MDR – H; wr; goto (MBR1)	Sottrae TOS da TOS Prelevato-1
iand1	MAR = SP = SP – 1; rd	Legge la parola sotto la cima dello stack
iand2	H = TOS	H = cima dello stack
iand3	MDR = TOS = MDR AND H; wr; goto (MBR1)	AND tra TOS Prelevato-1 e TOS
ior1	MAR = SP = SP – 1; rd	Legge la parola sotto la cima dello stack
ior2	H = TOS	H = cima dello stack
ior3	MDR = TOS = MDR OR H; wr; goto (MBR1)	OR tra TOS Prelevato-1 e TOS
dup1	MAR = SP = SP + 1	Incrementa SP e lo copia in MAR
dup2	MDR = TOS; wr; goto (MBR1)	Scrive la nuova parola sullo stack
pop1	MAR = SP = SP – 1; rd	Legge la parola sotto la cima dello stack
pop2		Attende che si completi la lettura
pop3	TOS = MDR; goto (MBR1)	Copia la nuova parola in TOS
swap1	MAR = SP – 1; rd	Legge la parola sotto la cima dello stack; imposta MAR a SP
swap2	MAR = SP	Prepara la scrittura della nuova seconda parola
swap3	H = MDR; wr	Salva il nuovo TOS; scrive la seconda parola nello stack
swap4	MDR = TOS	Copia il vecchio TOS in MDR
swap5	MAR = SP – 1; wr	Scrive il vecchio TOS nella seconda posizione dello stack
swap6	TOS = H; goto (MBR1)	Aggiorna TOS
bipush1	SP = MAR = SP + 1	Imposta MAR per scrivere la nuova cima dello stack
bipush2	MDR = TOS = MBR1; wr; goto (MBR1)	Aggiorna lo stack in TOS e nella memoria
iload1	MAR = LV + MBR1U; rd	Imposta MAR a LV + indice per la lettura
iload2	MAR = SP = SP + 1	SP punta alla nuova cima dello stack; prepara la scrittura
iload3	TOS = MDR; wr; goto (MBR1)	Aggiorna lo stack in TOS e nella memoria
istore1	MAR = LV + MBR1U	Imposta MAR a LV + indice
istore2	MDR = TOS; wr	Copia TOS per la scrittura in memoria
istore3	MAR = SP = SP – 1; rd	Decrementa SP, legge il nuovo TOS
istore4		Attende che si completi la lettura
istore5	TOS = MDR; goto (MBR1)	Aggiorna TOS
wide1	goto (MBR1 OR 0x100)	L'indirizzo successivo è il risultato dell'OR tra 0x100 e il codice operativo
wide_iload1	MAR = LV + MBR2U; rd; goto iload2	Identico a iload1, ma con un indice a 2 byte
wide_istore1	MAR = LV + MBR2U; goto istore2	Identico a istore1, ma con un indice a 2 byte
ldc_w1	MAR = CPP + MBR2U; rd; goto iload2	Uguale a wide_iload1, ma indicizza a partire da CPP

Figura 4.30 Microprogramma per Mic-2.

Etichetta	Operazioni	Commenti
iinc1	MAR = LV + MBR1U; rd	Imposta MAR a LV + indice per la lettura
iinc2	H = MBR1	Imposta H a una costante
iinc3	MDR = MDR + H; wr; goto (MBR1)	Incrementa di una costante e aggiorna
goto1	H = PC – 1	Copia PC in H
goto2	PC = H + MBR2	Aggiunge uno spiazzamento e aggiorna PC
goto3		Deve attendere che la IU prelevi il nuovo codice operativo
goto4	goto (MBR1)	Salto alla nuova istruzione
iflt1	MAR = SP = SP – 1; rd	Legge la parola sotto la cima dello stack
iflt2	OPC = TOS	Salva temporaneamente TOS in OPC
iflt3	TOS = MDR	Inserisce in TOS la nuova cima dello stack
iflt4	N = OPC; if (N) goto T; else goto F	Diramazione in base al bit N
gifeq1	MAR = SP = SP – 1; rd	Legge la parola sotto la cima dello stack
ifeq2	OPC = TOS	Salva temporaneamente TOS in OPC
ifeq3	TOS = MDR	Inserisce in TOS la nuova cima dello stack
ifeq4	Z = OPC; if (Z) goto T; else goto F	Diramazione in base al bit Z
if_icmpeq1	MAR = SP = SP – 1; rd	Legge la parola sotto la cima dello stack
if_icmpeq2	MAR = SP = SP – 1	Imposta MAR per leggere la nuova cima dello stack
if_icmpeq3	H = MDR; rd	Copia in H la parola sotto la cima dello stack
if_icmpeq4	OPC = TOS	Salva temporaneamente TOS in OPC
if_icmpeq5	TOS = MDR	Inserisce in TOS la nuova cima dello stack
if_icmpeq6	Z = H – OPC; if (Z) goto T; else goto F	Se le due parole in cima allo stack sono uguali, goto T, altrimenti, goto F
T	H = PC – 1; goto goto2	Uguale a goto1
F	H = MBR2	Aggiorna i byte in MBR2 per scartarne il contenuto
F2	goto (MBR1)	
invokevirtual1	MAR = CPP + MBR2U; rd	Inserisce in MAR l'indirizzo del puntatore al metodo
invokevirtual2	OPC = PC	Salva il PC di ritorno in OPC
invokevirtual3	PC = MDR	Imposta PC al primo byte del codice del metodo
invokevirtual4	TOS = SP – MBR2U	TOS = indirizzo di OBJREF – 1
invokevirtual5	TOS = MAR = H = TOS + 1	TOS = indirizzo di OBJREF
invokevirtual6	MDR = SP + MBR2U + 1; wr	Sovrascrive OBJREF con il puntatore di collegamento
invokevirtual7	MAR = SP = MDR	Imposta SP e MAR alla locazione in cui memorizzare il vecchio PC
invokevirtual8	MDR = OPC; wr	Si prepara a salvare il vecchio PC
invokevirtual9	MAR = SP = SP + 1	Incrementa SP per puntare alla locazione in cui memorizzare il vecchio LV
invokevirtual10	MDR = LV; wr	Salva il vecchio LV
invokevirtual11	LV = TOS; goto (MBR1)	Imposta LV per puntare al parametro Ø
ireturn1	MAR = SP = LV; rd	Reimposta SP e MAR per leggere il puntatore di collegamento
ireturn2		Attende il puntatore di collegamento
ireturn3	LV = MAR = MDR; rd	Imposta LV e MAR al puntatore di collegamento; legge il vecchio PC
ireturn4	MAR = LV + 1	Imposta MAR per puntare al vecchio LV; legge il vecchio LV
ireturn5	PC = MDR; rd	Ripristina PC
ireturn6	MAR = SP	
ireturn7	LV = MDR	Ripristina LV
ireturn8	MDR = TOS; wr; goto (MBR1)	Salva il valore di ritorno sulla cima originale dello stack

Figura 4.30 Microprogramma per Mic-2 (continua).

4.4.4 Architettura a pipeline: Mic-3

Mic-2 rappresenta ovviamente un miglioramento rispetto a Mic-1: è più veloce e utilizza una minor quantità di memoria di controllo, anche se in termini di "proprietà terriera" il costo della IFU è senza dubbio maggiore del guadagno ottenuto riducendo la memoria di controllo. Mic-2 è dunque una macchina considerevolmente più veloce con un prezzo leggermente maggiore.

Che cosa si può dire riguardo al tentativo di diminuire il tempo del ciclo? In buona parte il tempo del ciclo è determinato dalla tecnologia impiegata. Più piccoli sono i transistor e minore è la distanza che li separa, maggiore è la velocità alla quale può funzionare il clock. Per una data tecnologia il tempo richiesto per eseguire un'operazione sull'intero percorso dati è fisso (almeno dal nostro punto di vista). Ciononostante esistono alcuni spazi di libertà che tra poco cercheremo di sfruttare pienamente.

L'alternativa che abbiamo è quella di introdurre nella macchina un maggior parallelismo. Ora come ora Mic-2 funziona in modo altamente sequenziale; copia i registri sul bus, aspetta che la ALU e lo shifter li processino e infine scrive i risultati nuovamente all'interno dei registri. Fatta eccezione per la IFU, in questo funzionamento vi è poco parallelismo; vale la pena provare ad aumentarlo.

Com'è già stato detto, il ciclo del clock è limitato dal tempo necessario affinché i segnali si possano propagare lungo il percorso dati. La Figura 4.3 mostra schematicamente quali sono, durante ciascun ciclo, i ritardi attraverso i vari componenti. Il ciclo del percorso dati è composto da tre componenti principali:

1. il tempo necessario per portare i registri selezionati sui bus A e B
2. il tempo impiegato dalla ALU e dallo shifter per compiere il proprio lavoro
3. il tempo necessario per riportare i risultati nei registri e memorizzarli al loro interno.

Nella Figura 4.31 si vede una nuova architettura a tre bus che, oltre alla IFU, presenta anche tre *latch* (registri) inseriti a metà di ciascun bus. Questi registri vengono scritti a ogni ciclo. In effetti i latch spezzano il percorso dati in parti distinte che possono funzionare indipendentemente l'una dall'altra. Per riferirci a questa architettura utilizzeremo il nome **Mic-3**, oppure modello **a pipeline**.

Come possono essere d'aiuto questi registri? Con la nuova architettura occorrono ora tre cicli di clock per utilizzare il percorso dati: uno per caricare i latch A e B, uno per far funzionare la ALU e lo shifter e per caricare il latch C, e uno per memorizzare il latch nei registri. Siamo pazzi? Ovviamente no! Infatti il vantaggio dato dall'inserimento dei latch è duplice:

1. possiamo accelerare il clock dato che il ritardo massimo è ora più corto
2. possiamo usare tutte le parti del percorso dati durante ogni ciclo.

Spezzando il percorso dati in tre parti il ritardo massimo si riduce, con il risultato che la frequenza del clock può essere maggiore. Supponendo di dividere il ciclo del percorso dati in tre intervalli di tempo, lunghi circa 1/3 di quello originario, possiamo triplicare la velocità del clock. La cosa non è proprio realistica, dato che abbiamo anche aggiunto due registri all'interno del percorso dati, ma come prima approssimazione può essere accettabile.

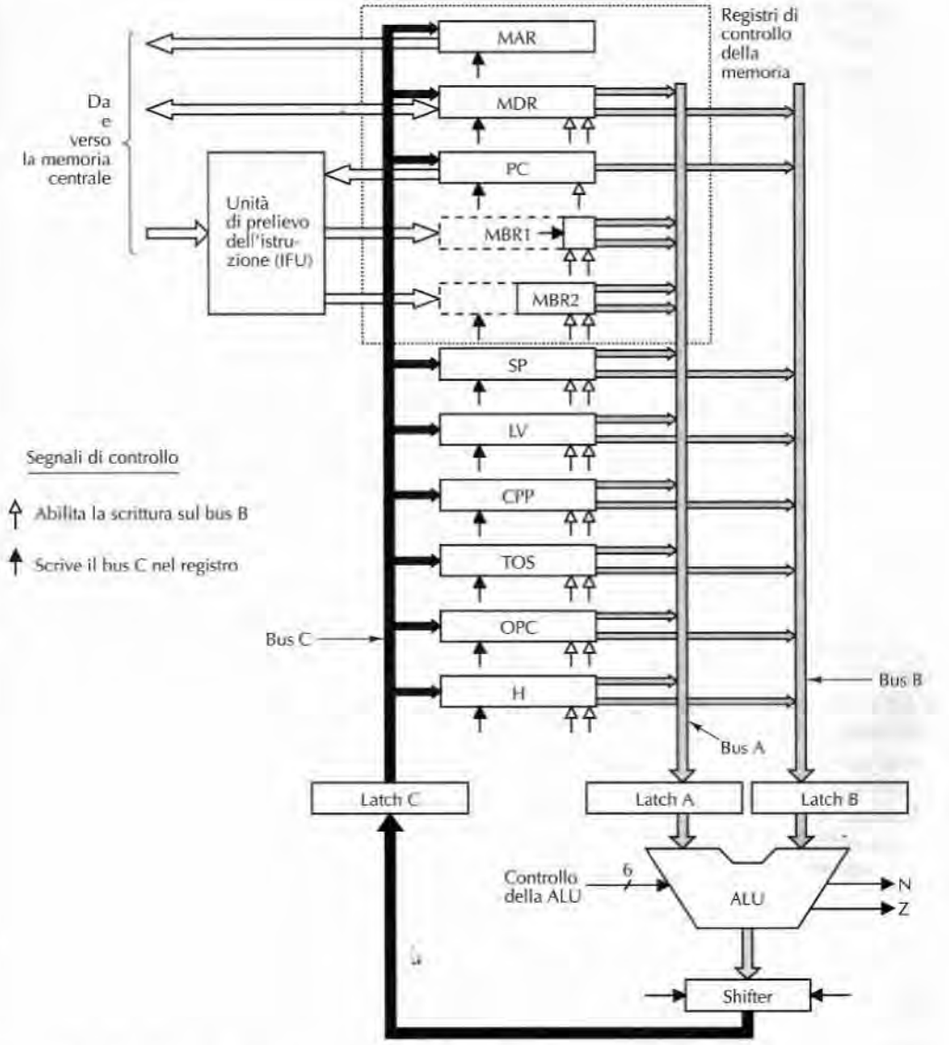


Figura 4.31 Percorso dati a tre bus utilizzato in Mic-3.

Dato che abbiamo assunto che letture e scritture di memoria possono essere soddisfatte dalla cache di primo livello e dato che questa cache è realizzata con lo stesso materiale dei registri, continueremo ad assumere che un'operazione di memoria richieda un solo ciclo (in pratica però ciò potrebbe non essere così semplice).

Il secondo punto riguarda il throughput piuttosto che la velocità delle singole istruzioni. In Mic-2 la ALU resta inattiva durante la prima e la terza parte di ogni ciclo di clock,

Etichetta	Operazioni	Commenti
swap1	MAR = SP – 1; rd	Legge la parola sotto la cima dello stack; imposta MAR a SP
swap2	MAR = SP	Si prepara a scrivere la nuova seconda parola
swap3	H = MDR; wr	Salva il nuovo TOS; scrive la seconda parola nello stack
swap4	MDR = TOS	Copia il vecchio TOS in MDR
swap5	MAR = SP – 1; wr	Scrive il vecchio TOS nella seconda posizione dello stack
swap6	TOS = H; goto (MBR1)	Aggiorna TOS

Figura 4.32 Codice di Mic-2 per SWAP.

ma spezzando il percorso dati in tre parti saremo in grado di utilizzare la ALU in ogni ciclo, ottenendo così una quantità di lavoro tre volte maggiore.

Poniamo ora la nostra attenzione sul percorso dati di Mic-3. Prima di iniziare ci serve però una notazione per i latch. La scelta più ovvia è quella di chiamarli A, B e C e di trattarli come dei registri, tenendo ben presenti i vincoli del percorso dati. La Figura 4.32 mostra la sequenza di codice relativa all'implementazione di `SWAP` per Mic-2.

Proviamo ora a implementare questa stessa sequenza sulla macchina Mic-3. Ricordiamoci che il percorso dati comprende ora tre cicli: uno per caricare `A` e `B`, uno per eseguire l'operazione e caricare `C` e uno per scrivere i risultati nei registri. Chiameremo **passo elementare** ciascuna di queste parti. L'implementazione di `SWAP` per Mic-3 è data nella Figura 4.33. Nel ciclo 1 iniziamo con `swap1` copiando `SP` in `B`. Non importa quale valore viene memorizzato in `A`, dato che per sottrarre 1 da `B` bisogna negare `ENA` (vedi Figura 4.2). Per semplicità non mostreremo gli assegnamenti che non sono utilizzati. Nel ciclo 2 effettuiamo la sottrazione. Nel ciclo 3 il risultato viene memorizzato in `MAR` e alla fine del ciclo inizia l'operazione di lettura (dopo che `MAR` è stato memorizzato). La lettura dalla memoria richiede un solo ciclo, e non sarà completata se non alla fine del ciclo 4; ciò è indicato dall'assegnamento a `MDR` durante il ciclo 4. Il valore in `MDR` non può essere letto prima del ciclo 5.

	Swap1	Swap2	Swap3	Swap4	Swap5	Swap6
C	MAR=SP–1; rd	MAR=SP	H=MDR; wr	MDR=TOS	MAR=SP–1; wr	TOS=H; goto (MBR1)
1	B=SP					
2	C=B–1	B=SP				
3	MAR=C; rd	C=B				
4	MDR=Mem	MAR=C				
5			B=MDR			
6			C=B	B=TOS		
7			H=C; wr	C=B	B=SP	
8			Mem=MDR	MDR=C	C=B–1	B=H
9					MAR=C; wr	C=B
10					Mem=MDR	TOS=C
11						goto (MBR1)

Figura 4.33 Implementazione di SWAP su Mic-3.

Tornando al ciclo 2, osserviamo che si può dapprima copiare `SP` in `B` per farlo passare all'interno della ALU nel ciclo 3 e infine memorizzarlo all'interno di `MAR` nel ciclo 4. Fin qui, nessun problema. Dovrebbe essere chiaro che se riusciamo a mantenere questo ritmo, iniziando una nuova microistruzione a ogni ciclo, potremo triplicare la velocità della macchina. Questo guadagno deriva dal fatto che possiamo lanciare a ogni ciclo una nuova microistruzione e che Mic-3 ha, rispetto a Mic-2, il triplo di cicli di clock per secondo. Quella che in realtà abbiamo creato è una CPU a pipeline.

Purtroppo nel ciclo 3 sorge un intoppo. Vorremmo cominciare a occuparci di `swap3`, ma la sua prima operazione consiste nel far passare `MDR` attraverso la ALU e `MDR` sarà disponibile dalla memoria solamente all'inizio del ciclo 5. La situazione in cui un passo elementare non può iniziare, dato che è in attesa di un risultato che non è ancora stato prodotto da un precedente passo elementare, viene chiamata **dipendenza RAW** (o **dipendenza effettiva**). Spesso ci si riferisce a queste situazioni con il termine *hazard*, cioè "rischio". RAW è l'acronimo di *Read After Write* ("lettura dopo scrittura") e indica che un passo elementare vuole leggere un registro che non è ancora stato scritto. In questo caso l'unica cosa sensata da fare è ritardare l'inizio di `swap3` finché `MDR` non sia disponibile, nel ciclo 5. Siamo cioè in una situazione di **stallo**. Una volta ottenuto il valore richiesto possiamo ricominciare a far partire una microistruzione per ciclo, dato che non ci sono più dipendenze. La microistruzione `swap6` sfiora una dipendenza, in quanto legge `H` nel ciclo immediatamente successivo a quello in cui `swap3` la scrive; se `swap5` avesse provato a leggere `H`, sarebbe rimasta in stallo per un ciclo.

Il programma di Mic-3 richiede più cicli di quello di Mic-2, ma viene eseguito più velocemente. Se indichiamo con ΔT ns il tempo di ciclo di Mic-3, allora Mic-3 impiega $11\Delta T$ ns per eseguire `SWAP`. Mic-2 impiega invece 6 cicli, ciascuno dei quali è lungo $3\Delta T$, per un totale di $18\Delta T$. L'uso della pipeline ha reso la macchina più veloce, anche se è necessario rimanere in stallo per risolvere una dipendenza.

Dato che l'uso della pipeline è una tecnica chiave in tutte le CPU moderne è importante comprenderne a fondo il funzionamento. Nella Figura 4.34 vediamo il percorso dati della Figura 4.31 reso graficamente come una pipeline. La prima colonna rappresenta ciò che avviene durante il primo ciclo, la seconda colonna ciò che avviene nel secondo, e così via (assumendo che non si verifichino stalli). La regione in grigio dell'istruzione 1 durante il ciclo 1 indica che la IFU è occupata nel prelievo dell'istruzione 1. Al successivo scoccare del clock, durante il ciclo 2, i registri richiesti dall'istruzione 1 vengono caricati nei latch A e B, mentre la IFU è occupata nel prelievo dell'istruzione 2; anche in questo caso le operazioni sono indicate da rettangoli grigi.

Durante il ciclo 3 l'istruzione 1 utilizza la ALU e lo shifter per eseguire la propria operazione, mentre i latch A e B vengono caricati per l'istruzione 2 e, allo stesso tempo, l'istruzione 3 comincia a essere prelevata. Infine, durante il ciclo 4, ci sono quattro istruzioni in corso d'esecuzione. I risultati dell'istruzione 1 cominciano a essere memorizzati, il lavoro della ALU per l'istruzione 2 volge al termine, i latch A e B vengono caricati per l'istruzione 3 e si comincia a prelevare l'istruzione 4.

Se avessimo mostrato il ciclo 5 e quelli successivi la situazione sarebbe stata la stessa del ciclo 4: tutte e quattro le parti del percorso dati che possono funzionare indipendentemente continuano infatti a lavorare in parallelo. Questa organizzazione rappresenta una pipeline a 4 stadi, in cui uno preleva le istruzioni, uno accede agli operandi, uno esegue le

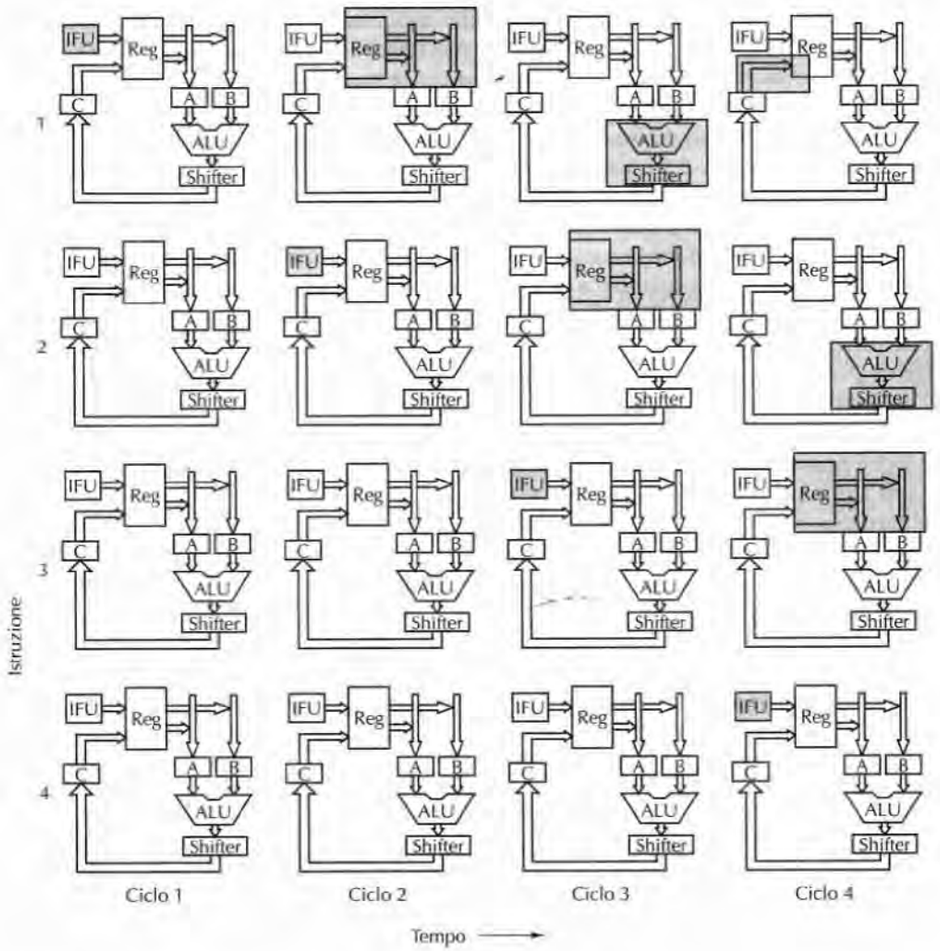


Figura 4.34 Rappresentazione grafica del funzionamento di una pipeline.

operazioni della ALU e uno scrive i risultati nei registri. La pipeline è simile a quella della Figura 2.4(a), con la differenza che manca lo stadio di decodifica. Nell'uso della pipeline l'aspetto più importante da cogliere è che, anche se una singola istruzione richiede quattro cicli di clock per essere eseguita, durante ogni ciclo di clock può iniziare una nuova istruzione e mentre una, iniziata precedente, termina la propria esecuzione.

La Figura 4.34 può essere guardata in un altro modo, seguendo orizzontalmente ciascuna istruzione lungo tutta la larghezza della pagina. Per quanto riguarda l'istruzione 1, durante il ciclo 1 la IFU è in funzione. Nel ciclo 2 i suoi registri cominciano a essere portati sui bus A e B. Nel ciclo 3 la ALU e lo shifter svolgono le operazioni da lei richieste. Infine, nel ciclo 4, i suoi risultati sono memorizzati nuovamente all'interno dei registri. Occorre notare che sono

disponibili quattro sezioni dell'hardware e durante ciascun ciclo una data istruzione ne utilizza uno soltanto, lasciando le restanti sezioni a disposizione delle altre istruzioni.

Un'analogia utile a capire come sono organizzate le pipeline è la linea di montaggio di una fabbrica di automobili. Per astrarre gli aspetti essenziali del modello immaginiamo che a ogni minuto venga colpito un grande gong e che in quel momento tutte le macchine si spostino, lungo la linea di montaggio, da un'isola a quella successiva. In ogni stazione i lavoratori eseguono una particolare operazione sull'automobile che hanno di fronte, per esempio aggiungere il volante o installare i freni. A ciascun battito del gong (1 ciclo) viene inserita una nuova automobile nella linea di montaggio e un'altra termina di essere assemblata. Quindi a ogni ciclo viene completata una macchina, anche se l'intero assemblaggio potrebbe constare di centinaia di cicli. La fabbrica può produrre un'automobile al minuto indipendentemente da quanto tempo è effettivamente necessario per assemblare un'automobile. Questa è la potenza della pipeline, e può essere sfruttata tanto per le CPU quanto per le fabbriche di automobili.

4.4.5 Pipeline a sette stadi: Mic-4

Un aspetto sul quale abbiamo intenzionalmente sorvolato riguarda il fatto che ciascuna microistruzione selezioni il proprio successore. La maggior parte di loro seleziona semplicemente quella che la segue nella sequenza corrente, mentre l'ultima, come `swap6`, spesso esegue una diramazione che blocca la pipeline, dato che è impossibile continuare a prelevare nuove microistruzioni. Occorre trovare un modo migliore per gestire questo punto.

La nostra successiva (e ultima) microarchitettura è Mic-4. Le sue parti principali sono mostrate nella Figura 4.35, nella quale sono stati trascurati molti dettagli per rendere più chiara l'illustrazione. Come per Mic-3, anche questa microarchitettura è dotata di una IFU che preleva dalla memoria le parole e mantiene i vari registri `MBR`.

La IFU alimenta inoltre con il flusso di byte una nuova componente, chiamata **unità di decodifica**. Questa unità è dotata di una ROM interna indicizzata attraverso il codice operativo IJVM. Ciascuna riga contiene due parti: la lunghezza dell'istruzione IJVM corrispondente e un indice relativo a un'altra ROM: la ROM delle micro-operazioni. La lunghezza dell'istruzione IJVM è utilizzata per permettere all'unità di decodifica di analizzare il flusso di byte entrante e suddividerlo in istruzioni, sapendo in ogni momento quali byte rappresentano i codici operativi e quali invece gli operandi. Se l'istruzione corrente è lunga 1 byte (per esempio, `POP`), allora l'unità di decodifica sa che il byte successivo è un codice operativo. Se invece l'istruzione corrente è lunga 2 byte, l'unità di decodifica sa che il byte successivo è un operando, seguito da un altro codice operativo. Quando incontra il prefisso `WIDE`, l'unità trasforma il byte successivo in uno speciale codice operativo più ampio (per esempio `WIDE + ILOAD` diventa `WIDE_ILOAD`).

L'unità di decodifica invia alla componente successiva, chiamata **unità di accodamento**, l'indice relativo alla ROM delle micro-operazioni che ha trovato nella sua tabella. Oltre a vari circuiti, quest'unità contiene due tabelle interne, una in una RAM e l'altra in una ROM. Quest'ultima tabella contiene il microprogramma, in cui ciascuna istruzione IJVM ha un certo numero di elementi consecutivi, chiamati **micro-operazioni**. Queste devono essere in ordine e quindi non è possibile usare espedienti come quello che in Mic-2 permette a `wide_iload2` di effettuare un salto verso `iload2`. Ogni sequenza IJVM deve essere scandita interamente. Per tale motivo, alcune sequenze devono essere duplicate.

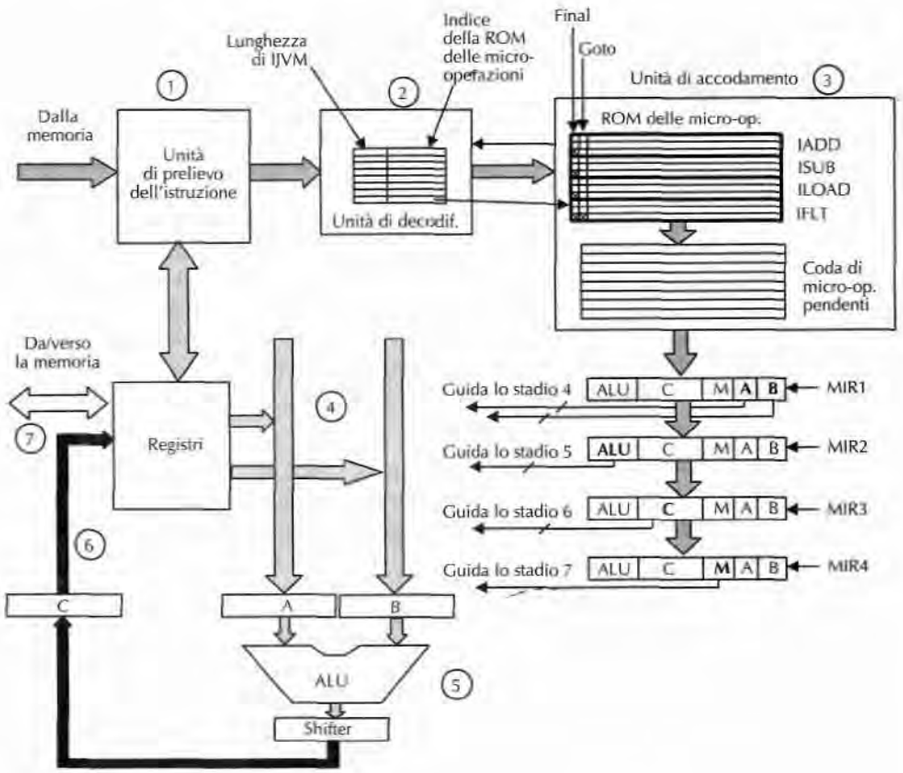


Figura 4.35 Componenti principali di Mic-4.

Le micro-operazioni sono simili alle microistruzioni della Figura 4.5, tranne per il fatto che sono prive dei campi `NEXT_ADDRESS` e `JAM`, ma hanno un nuovo campo per specificare l'input del bus A. Sono anche presenti due nuovi bit: Final e Goto. Il bit Final viene impostato a 1 nell'ultima micro-operazione IJVM di ciascuna sequenza per indicarne la fine. Il bit Goto viene invece impostato a 1 per indicare le micro-operazioni che effettuano microdiramazioni condizionali; il loro formato è diverso da quello delle normali micro-operazioni ed è costituito dai bit `JAM` e da un indice relativo alla ROM delle micro-operazioni. Le microistruzioni, che precedentemente eseguivano delle operazioni con il percorso dati ed effettuavano allo stesso tempo una microdiramazione condizionale (per esempio, `iflt4`), devono ora essere separate in due micro-operazioni distinte.

L'unità di accodamento funziona nel modo seguente. Riceve dall'unità di decodifica un indice della ROM delle micro-operazioni, cerca la micro-operzione corrispondente e la copia in una coda interna. Successivamente copia nella coda anche le due micro-operazioni seguenti. L'unità continua in questo modo finché non incontra una micro-operazione in cui il bit Final vale 1; in questo caso la copia e si arresta. L'unità di accodamento, assumendo che non abbia incontrato una micro-operazione con il bit Goto attivo e che ci sia ancora

molto spazio nella coda, rispedisce all'unità di decodifica un segnale di conferma. Quando vede la conferma, l'unità di decodifica spedisce all'unità di accodamento l'indice della successiva istruzione IJVM.

In questo modo la sequenza d'istruzioni IJVM in memoria viene convertita in una sequenza di micro-operazioni in una coda. Queste micro-operazioni alimentano i registri `MIR`, che spediscono i segnali che permettono di controllare il percorso dati. Esiste tuttavia un altro fattore che dobbiamo considerare: i campi delle micro-operazioni non sono attivi nello stesso momento. I campi `A` e `B` sono attivi durante il secondo ciclo, il campo `C` lo è durante il terzo ciclo e ogni operazione di memoria si svolge nel quarto ciclo.

Per permettere che ciò funzioni correttamente abbiamo introdotto nella Figura 4.35 quattro registri `MIR` indipendenti. All'inizio di ciascun ciclo di clock (il tempo Δw nella Figura 4.3) `MIR3` viene copiato in `MIR4`, `MIR2` in `MIR3`, `MIR1` in `MIR2` e `MIR1` viene caricato con una nuova micro-operazione proveniente dalla coda delle micro-operazioni. Ogni `MIR` emette i propri segnali, ma non tutti vengono utilizzati. I campi `A` e `B` provenienti da `MIR1` selezionano i registri che guidano i latch A e B, mentre il campo `ALU` in `MIR1` non viene utilizzato e non è connesso a nessuna componente del percorso dati.

Dopo un ciclo di clock, questa micro-operazione si è spostata in `MIR2` e i registri che ha selezionato sono stati salvati nei latch A e B, in attesa che l'avventura cominci. Il suo campo `ALU` viene ora utilizzato per guidare la ALU. Nel ciclo successivo il suo campo `C` riscriverà il risultato nei registri e poi si sposterà in `MIR4` e darà inizio all'operazione di memoria richiesta utilizzando `MAR` (o `MDR`, nel caso di una scrittura), che sarà già stato caricato.

C'è un ultimo aspetto di Mic-4 che occorre approfondire: le microdiramazioni. Alcune istruzioni IJVM, come `IFLT`, richiedono di effettuare dei salti condizionati in base, per esempio, al bit `N`. Quando si verifica una microdiramazione la pipeline non può continuare. Per gestire questo caso abbiamo aggiunto alla micro-operazione il bit Goto. Quando incontra una micro-operazione che possiede questo bit attivo mentre la sta copiando all'interno della coda, l'unità di accodamento capisce in anticipo che ci sarà un problema e non spedisce all'unità di decodifica il segnale di conferma. Il risultato è che la macchina rimarrà in stallo in questo punto finché la microdiramazione non sarà stata risolta.

Presumibilmente alcune istruzioni IJVM successive alla diramazione saranno già state portate nell'unità di decodifica (ma non nell'unità di accodamento) nel momento in cui si incontra una micro-operazione con il bit Goto attivo e non viene rispedito il segnale di conferma (cioè il permesso per continuare). Per riportare un po' d'ordine nella situazione e ritornare sui propri passi è necessario disporre di hardware speciale; la spiegazione di questi meccanismi esula però dagli scopi del libro. Edsger Dijkstra non ebbe idea di quanto avesse ragione quando scrisse il famoso: "GOTO Statement Considered Harmful" (Dijkstra, 1968a), cioè "L'istruzione GOTO è da considerarsi pericolosa".

Abbiamo percorso un lungo cammino a partire da Mic-1. Mic-1 è un componente hardware molto semplice, in cui quasi tutto il controllo è implementato in software. Mic-4 è organizzata altamente a pipeline, con sette stadi e un hardware molto più complesso. La pipeline è mostrata schematicamente nella Figura 4.36, in cui i numeri cerchiati fanno riferimento ai componenti della Figura 4.35. Mic-4 effettua automaticamente il prefetch di un flusso di byte dalla memoria, li decodifica trasformandoli in istruzioni IJVM che converte in una sequenza di micro-operazioni utilizzando una ROM; accoda poi queste micro-operazioni per poterle utilizzare quando sarà necessario. Se lo si desidera è possibile legare i

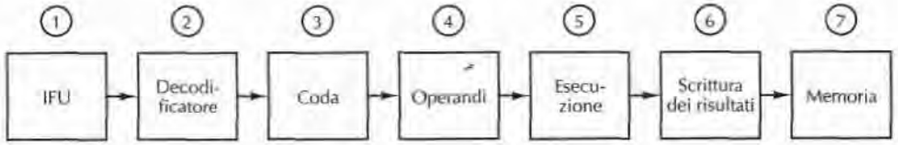


Figura 4.36 Pipeline di Mic-4.

primi tre stadi della pipeline al clock del percorso dati, ma in realtà non c'è sempre del lavoro da eseguire. La IFU, per esempio, può certamente evitare di inviare durante ogni ciclo di clock un nuovo codice operativo IJVM all'unità di decodifica; le istruzioni IJVM richiedono infatti più cicli per essere eseguite e se si spedisse un codice operativo a ogni ciclo si riempirebbe rapidamente la coda.

A ogni ciclo di clock vengono traslati in avanti i registri `MIR` e la micro-operazione che si trova alla fine della coda viene copiata in `MIR1` per iniziare la propria esecuzione. I segnali di controllo che partono dai quattro `MIR` si diffondono dunque lungo il percorso dati generando le azioni desiderate. Ciascun `MIR` controlla una parte diversa del percorso dati e quindi passi elementari differenti.

In questa architettura abbiamo una CPU organizzata profondamente a pipeline, con passi molto brevi e quindi una frequenza di clock molto elevata. Molte CPU sono progettate in questo modo, specialmente quelle che devono implementare un vecchio (CISC) insieme d'istruzioni. Per esempio, come vedremo più avanti nel corso di questo capitolo, l'implementazione del Pentium II è, per certi aspetti, concettualmente simile a Mic-4.

4.5 Miglioramento delle prestazioni

Tutti i produttori di calcolatori vorrebbero che le loro macchine funzionassero alla massima velocità possibile. In questo paragrafo presenteremo alcune tecniche avanzate sulle quali si sta focalizzando la ricerca attuale al fine di migliorare le prestazioni dei sistemi (principalmente quelle della CPU e della memoria). Per via della natura fortemente competitiva del mercato dei calcolatori l'intervallo di tempo che passa dal momento in cui la ricerca propone nuove idee per rendere più veloce un computer e la loro effettiva implementazione in prodotti commerciali è estremamente breve. Per questo motivo la maggior parte delle idee che tratteremo sono già in uso in un'ampia gamma di prodotti disponibili sul mercato.

È possibile dividere l'esposizione in due categorie principali: i miglioramenti dell'implementazione e quelli dell'architettura. I primi sono i modi di realizzare nuove CPU o nuove memorie che permettono al sistema di essere più veloce senza modificarne l'architettura. Ciò significa che i vecchi programmi potranno essere eseguiti anche sulla nuova macchina, aspetto che rappresenta un grande vantaggio per il successo commerciale del prodotto. Un modo, ma non l'unico, per migliorare l'implementazione è l'utilizzo di un clock più veloce. I guadagni di prestazioni ottenuti dall'80386, passando per l'80486, il Pentium e il Pentium Pro, fino ad arrivare al Pentium II, sono dovuti a una migliore implementazione, mentre l'architettura è rimasta essenzialmente la stessa per tutte le macchine.

Alcuni tipi di miglioramenti possono essere ottenuti solamente cambiando l'architettura. A volte questi cambiamenti sono incrementali e consistono per esempio nell'aggiunta di nuove istruzioni o registri, in modo che i vecchi programmi possano continuare a funzionare anche sui nuovi modelli. In questo caso, per trarre pieno vantaggio dalle prestazioni della macchina, è necessario modificare il software o almeno ricompilarlo con un nuovo compilatore che tenga conto delle nuove funzionalità.

A volte i progettisti si rendono tuttavia conto che la vecchia architettura, essendo sopravvissuta fin troppo a lungo, ha esaurito la propria utilità e che l'unico modo per ottenere dei progressi è quello di ricominciare da capo progettandone una nuova. La rivoluzione RISC negli anni '80 fu uno di questi punti di rottura; un altro sta per arrivare, e nel corso del Capitolo 5 ne vedremo un esempio (l'Intel IA-64).

Nel resto del paragrafo analizzeremo quattro diverse tecniche per migliorare le prestazioni della CPU. Inizieremo presentando tre miglioramenti la cui implementazione si è ormai affermata nelle macchine attuali e successivamente passeremo a un miglioramento che, per funzionare al massimo, necessita di un piccolo supporto da parte dell'architettura. Queste tecniche sono la memoria cache, la predizione dei salti, l'esecuzione fuori sequenza con la rinomina dei registri e l'esecuzione speculativa.

4.5.1 Memoria cache

Nella storia della progettazione dei calcolatori una delle sfide più ardue è stata quella di definire un sistema di memoria capace di fornire al processore gli operandi alla velocità alla quale esso li può elaborare. Il recente ed elevato tasso di crescita della velocità dei processori non è stato accompagnato da un analogo incremento della velocità delle memorie. Negli ultimi decenni le memorie sono diventate più lente rispetto alle CPU. Per via dell'enorme importanza che riveste la memoria primaria questa situazione ha frenato in modo significativo lo sviluppo di sistemi ad alte prestazioni, e allo stesso tempo ha stimolato la ricerca di nuovi modi per aggirare il problema della velocità della memoria, che diventa di anno in anno più lenta rispetto alla CPU.

I processori moderni effettuano una quantità travolgente di richieste al sistema di memoria, sia in termini di latenza (il ritardo nel fornire un operando) sia in termini di banda (la quantità di dati fornita per unità di tempo). Questi due aspetti che caratterizzano un sistema di memoria sono purtroppo in larga parte in conflitto. Molte tecniche che permettono di aumentare la larghezza di banda incrementano allo stesso tempo la latenza. Per esempio, la pipeline utilizzata in Mic-3 può essere applicata a un sistema di memoria in modo da gestire efficientemente la sovrapposizione di più richieste alla memoria. Purtroppo, come per Mic-3, si ottiene una latenza maggiore per le singole operazioni di memoria. Dato che la velocità di clock del processore continua ad aumentare, diventa sempre più difficile realizzare un sistema di memoria in grado di fornire gli operandi in uno o due cicli di clock.

Un modo per combattere questo problema è l'uso delle cache. Come abbiamo visto nel Paragrafo 2.2.5 le cache mantengono all'interno di una piccola e veloce memoria le parole di memoria utilizzate più di recente, in modo da accelerarne l'accesso. Se nella cache è presente una percentuale sufficientemente elevata delle parole di memoria necessarie, è possibile ridurre enormemente l'effettiva latenza della memoria.

Una delle tecniche più efficaci per migliorare sia la larghezza di banda sia la latenza è l'uso di più cache. Una tecnica elementare che funziona in modo efficace consiste nell'in-

trodurre due cache separate, una per le istruzioni e una per i dati. Il sistema con due cache distinte per i dati e per le istruzioni, spesso chiamato **a cache separata**, fornisce vari vantaggi. In primo luogo è possibile far partire le operazioni di memoria indipendentemente per ciascuna cache, raddoppiando effettivamente la larghezza di banda del sistema di memoria. Questo è il motivo per cui ha senso fornire due porte distinte per la memoria, come abbiamo fatto in Mic-1: ogni porta ha la propria cache. Occorre notare che le due cache hanno accessi indipendenti alla memoria centrale.

Attualmente molti sistemi di memoria sono ancora più complicati. Spesso tra la memoria centrale e le cache delle istruzioni e dei dati è presente un'altra cache, chiamata **cache di secondo livello**. In realtà, per disporre di sistemi di memoria ancora più sofisticati, si possono avere anche tre o più livelli di cache. Il chip stesso della CPU contiene una piccola cache per le istruzioni e una piccola cache per i dati, le cui dimensioni sono in genere comprese tra 16 e 64 KB. Dopo questa piccola memoria c'è una cache di secondo livello, che non si trova nel chip della CPU, ma che può essere inclusa all'interno del suo involucro, vicina al chip e a esso collegata tramite circuiti ad alta velocità. Questa cache è di solito unificata e contiene quindi sia dati sia istruzioni. Generalmente la dimensione delle cache L2 è compresa tra 512 KB e 1 MB. La cache di terzo livello si trova sulla scheda del processore ed è costituita da alcuni megabyte di SRAM, un tipo di memoria molto più veloce rispetto alla memoria DRAM centrale. Di solito le cache sono annidate, nel senso che l'intero contenuto della cache di primo livello è compreso nella cache di secondo livello e tutto il contenuto di quest'ultima è compreso in quella di terzo livello.

Per raggiungere il proprio scopo le cache sfruttano due tipi di località degli indirizzi. La **località spaziale** riflette la considerazione che è molto probabile che in un prossimo futuro si accederà a locazioni di memoria i cui indirizzi sono numericamente simili a quello di una locazione appena utilizzata. La **località temporale** si verifica quando si accede nuovamente a locazioni di memoria già utilizzate di recente. Ciò potrebbe avvenire per esempio nel caso di locazioni di memoria vicine alla cima dello stack oppure nel caso delle istruzioni di un ciclo. Nella progettazione di una cache la località temporale viene principalmente

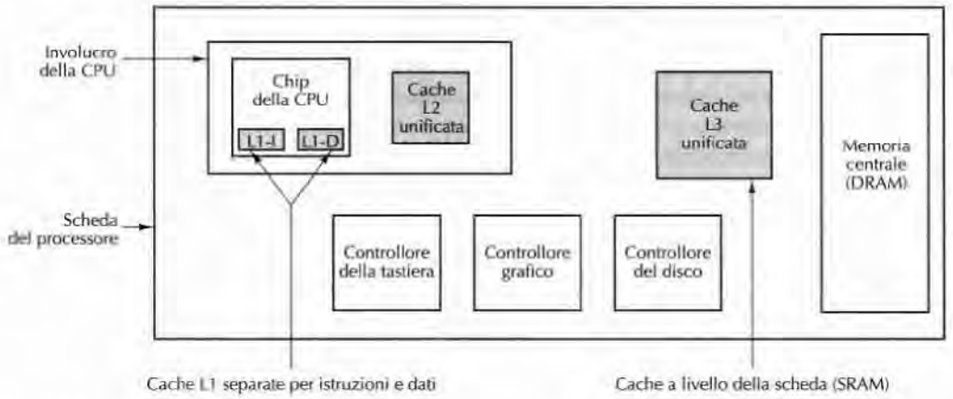


Figura 4.37 Sistema con tre livelli di cache.

sfruttata scegliendo che cosa scartare nel caso in cui si verifichi un fallimento della cache. Molti algoritmi di sostituzione sfruttano la località temporale scartando gli elementi che non sono stati utilizzati di recente.

Tutte le cache utilizzano lo stesso modello. La memoria centrale è divisa in blocchi di dimensione fissa chiamati **linee di cache**. Una linea di cache è composta generalmente da 4 a 64 byte consecutivi. Le linee sono numerate consecutivamente a partire da 0: se la dimensione della linea è di 32 byte, la linea 0 conterrà i byte compresi tra 0 e 31, la linea 1 quelli compresi tra 32 e 63, e così via. In ogni momento la cache contiene delle linee. Quando si effettua un riferimento alla memoria il circuito che controlla la cache verifica se la parola richiesta si trova nella cache. Se così è, la parola può essere utilizzata, risparmiando un viaggio fino alla memoria centrale. In caso contrario, viene rimossa una linea dalla cache per sostituirla con la linea richiesta, prelevata dalla memoria centrale oppure da un livello di cache inferiore. Esistono molte varianti di questo schema, ma in tutte l'idea base è di tenere il più a lungo possibile nella cache le linee usate più frequentemente, in modo da massimizzare il numero di riferimenti alla memoria che la cache riesce a soddisfare.

Cache a corrispondenza diretta

La cache più semplice è conosciuta con il nome di **cache a corrispondenza diretta** (*direct-mapped cache*). La Figura 4.38(a) mostra un esempio di cache di questo tipo che contiene 2048 elementi. Ciascun elemento (riga) della cache può memorizzare esattamente una linea di cache della memoria centrale. Se ipotizziamo che la linea di cache abbia dimensione di 32 byte, la cache può memorizzare 64 KB. Gli elementi della cache sono composti da tre parti.

1. Il bit `Valid` che indica se il dato nell'elemento è valido oppure no. All'avvio del sistema tutti gli elementi della cache sono marcati come non validi.
2. Il campo `Tag` che è un valore univoco a 16 bit, corrispondente alla linea di memoria da cui provengono i dati.
3. Il campo `Data` che contiene una copia del dato della memoria. Questo campo memorizza una linea di cache di 32 byte.

Nelle cache a corrispondenza diretta una data parola di memoria può essere memorizzata in un'unica posizione della cache. Per ogni indirizzo di memoria esiste un solo posto all'interno della cache in cui cercare il dato. Per memorizzare e prelevare dati dalla cache l'indirizzo è diviso in quattro campi, come mostra la Figura 4.38(b):

1. il campo `TAG` corrisponde ai bit `Tag` memorizzati in un elemento della cache
2. il campo `LINE` indica l'elemento della cache contenente i dati corrispondenti, se sono presenti
3. il campo `WORD` indica a quale parola si fa riferimento all'interno della linea
4. il campo `BYTE` non viene generalmente utilizzato, ma se si richiede un solo byte esso indica quale byte è richiesto all'interno della parola. Per una cache che fornisce soltanto parole a 32 bit, questo campo varrà sempre 0.

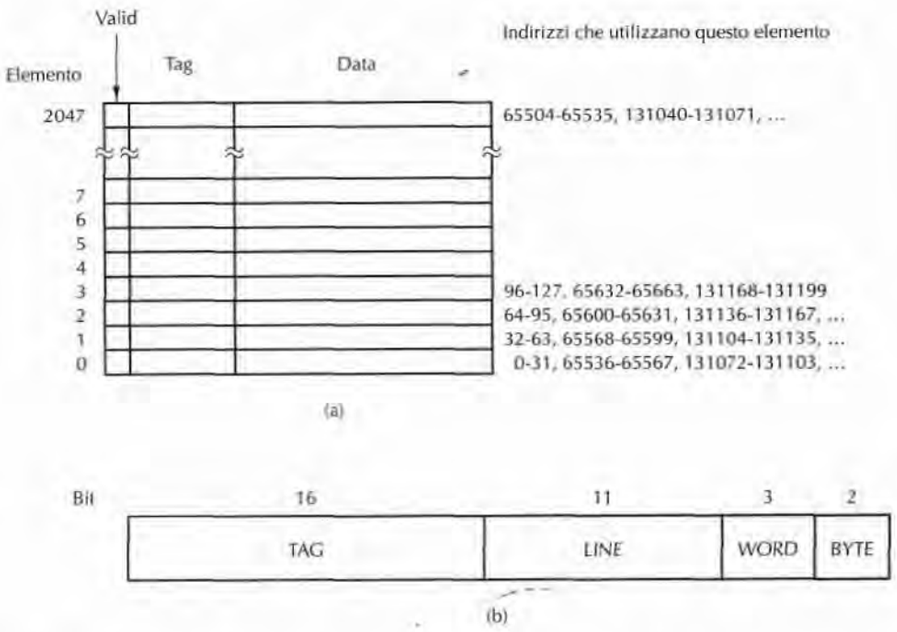


Figura 4.38 (a) Cache a corrispondenza diretta. (b) Indirizzo virtuale a 32 bit.

Quando la CPU genera un indirizzo di memoria, l'hardware estrae dall'indirizzo gli 11 bit del campo `LINE` e li usa come indice all'interno della cache per cercare uno dei 2048 elementi. Se questo elemento è valido si effettua un confronto tra il campo `TAG` dell'indirizzo di memoria e il campo `Tag` dell'elemento della cache. Se sono uguali, l'elemento della cache contiene la parola richiesta; questa situazione è chiamata **cache *bit*** ("successo della cache"). In tal caso la parola che si vuole leggere può essere presa direttamente dalla cache, evitando di dover andare fino alla memoria. Dalla linea di cache viene estratta solamente la parola effettivamente richiesta, mentre non viene utilizzato il resto della linea. Se l'elemento della cache non è valido oppure se i due campi "tag" non corrispondono, il dato richiesto non è presente nella cache: questa situazione è chiamata **cache *miss*** ("fallimento della cache"). In questo caso la linea della cache a 32 byte viene prelevata dalla memoria e memorizzata nell'elemento appropriato della cache, sostituendo quello che era presente precedentemente. In ogni caso se l'elemento della cache era stato modificato dopo averlo caricato, occorre riscriverlo in memoria prima di poterlo scartare.

Nonostante la complessità della decisione, l'accesso alla parola richiesta può essere notevolmente veloce. Nel momento in cui si conosce l'indirizzo, si conosce anche l'esatta locazione della parola *sempre che si trovi nella cache*. Ciò significa che è possibile leggere la parola e spedirla al processore nello stesso momento in cui esso sta controllando se la parola è quella corretta (confrontando i due "tag"). Il processore riceve quindi dalla cache una parola nello stesso momento (se non addirittura in anticipo) in cui viene a sapere se la parola è effettivamente quella richiesta.

Questo schema di corrispondenza inserisce linee di memoria consecutive in elementi della cache consecutivi. Nella cache è possibile memorizzare effettivamente fino a 64 KB di dati contigui. Tuttavia all'interno della cache non è possibile memorizzare allo stesso tempo due linee i cui indirizzi differiscono esattamente di 64 KB (65.536 byte) o di un multiplo di questo valore (dato che il loro campo `LINE` è uguale). Per esempio, se un programma accede a dati che si trovano nella locazione X e successivamente esegue un'istruzione che richiede dati presenti alla locazione $X + 65.536$ (o in qualsiasi altra locazione della stessa linea), la seconda istruzione obbligherà a ricaricare l'elemento della cache, sovrascrivendo quello precedente. Se ciò avviene con una certa frequenza, le prestazioni degradano. Il comportamento di una cache può essere addirittura peggiore di quanto non sarebbe in assenza di cache, dato che ogni operazione di memoria comporta la lettura di un'intera linea di cache e non semplicemente di una parola.

Le cache a corrispondenza diretta sono le più comuni e funzionano in modo efficiente, dato che collisioni come quelle descritte precedentemente si verificano molto raramente, se non addirittura mai. Un compilatore molto astuto, nel posizionare i dati e le istruzioni all'interno della memoria, potrebbe per esempio tener conto delle possibili collisioni della cache, cercando di evitarle. Si noti che il caso particolare descritto precedentemente non si potrebbe verificare in un sistema con cache separate per le istruzioni e per i dati, in quanto le richieste che erano in collisione verrebbero in questo caso gestite da cache differenti. Utilizzare due cache al posto di una sola porta un secondo vantaggio: una maggiore flessibilità nella gestione d'indirizzi di memoria in conflitto.

Cache set-associative

Com'è stato detto precedentemente, molte linee della memoria sono in competizione per l'utilizzo di uno stesso elemento della cache. Se un programma che utilizza la cache della Figura 4.38(a) usa intensamente le parole comprese tra l'indirizzo 0 e l'indirizzo 65.536, si verificheranno costantemente dei conflitti e ogni riferimento alla memoria potrebbe potenzialmente espellere un elemento dalla cache. Una soluzione a questo problema è quello di consentire che in ciascun elemento della cache ci possano essere due o più linee. Una cache con n possibili elementi per ciascun indirizzo è chiamata **cache set-associativa a *n* vie**. La Figura 4.39 mostra una cache associativa a quattro vie.

Queste cache sono più complesse di quelle a corrispondenza diretta. Infatti, anche se l'elemento della cache da esaminare viene calcolato a partire dall'indirizzo di memoria, è tuttavia necessario controllare un insieme di n elementi per vedere se la linea richiesta è presente nella cache. L'esperienza ha mostrato tuttavia che cache a due o a quattro vie garantiscono prestazioni sufficientemente buone, da rendere accettabile l'aumento dei componenti digitali richiesti.

L'uso di una cache set-associativa pone il progettista di fronte a una scelta; quando viene portata all'interno della cache una nuova linea, quale delle presenti deve essere scartata? Per poter compiere la scelta migliore bisognerebbe poter predire il futuro; un algoritmo che nella maggior parte delle situazioni funziona abbastanza bene è detto **LRU**, acronimo di *Last Recently Used*, cioè "meno recentemente usata". Questo algoritmo mantiene un ordinamento temporale di ciascun insieme di elementi ai quali si può far riferimento in base a una certa locazione di memoria. Quando si accede a una delle linee presenti nella cache l'algoritmo aggiorna la lista marcando l'elemento utilizzato più di recente. Quando giunge il momento

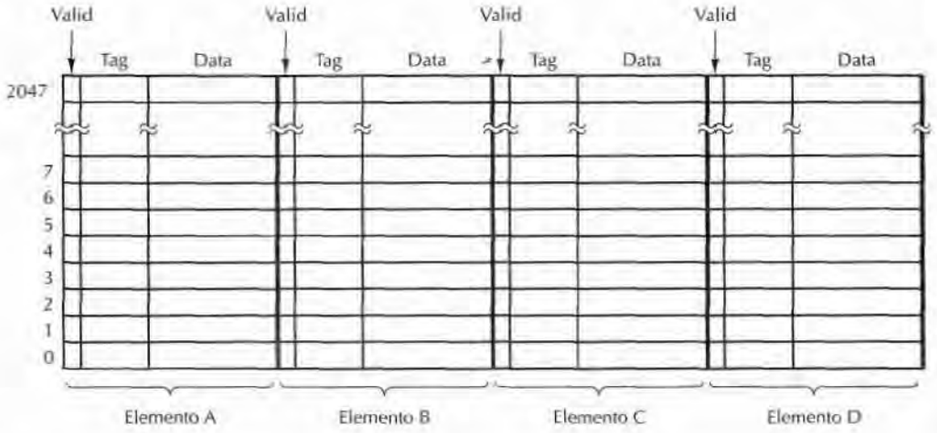


Figura 4.39 Cache associativa a quattro vie.

di sostituire un elemento, l'algoritmo scarta quello che si trova alla fine della lista, cioè quello al quale è stato effettuato un accesso meno di recente.

Portando l'idea all'estremo è possibile immaginare una cache a 2048 vie contenente un unico insieme di 2048 linee. In questo caso tutti gli indirizzi di memoria vengono mappati in un unico insieme e quindi la ricerca all'interno della cache obbliga a confrontare l'indirizzo con 2048 diversi tag. Occorre notare che ciascun elemento deve avere dei componenti logici che permettano di controllare la corrispondenza con il tag. Dato che la lunghezza del campo `LINE` è pari a 0, il campo `TAG` costituisce l'intero indirizzo, fatta eccezione per i campi `WORD` e `BYTE`. Inoltre quando viene rimpiazzata una linea della cache, tutte e 2048 le locazioni sono candidate alla sostituzione. Per mantenere e gestire una lista di 2048 elementi è necessario un lavoro talmente grande da rendere inapplicabile l'algoritmo LRU. (Ricordiamoci che questa lista deve essere aggiornata per ogni operazione di memoria, non soltanto quando si verifica un fallimento della cache.) Nella maggior parte delle circostanze le cache set-associative a molte vie non hanno prestazioni sensibilmente migliori di quelle con un piccolo numero di vie. Il risultato sorprendente è che in alcuni casi le prestazioni sono addirittura peggiori. Per questi motivi è raro incontrare cache con più di quattro vie.

Infine, anche le operazioni di scrittura pongono un particolare problema nell'uso delle cache. Quando un processore scrive una parola, e questa parola è presente nella cache, ovviamente esso deve aggiornare la parola oppure scaricare l'elemento dalla cache. In quasi tutte le progettazioni si sceglie di aggiornare la cache. Ma che cosa si può dire riguardo l'aggiornamento della copia nella memoria centrale? Questa operazione può essere rinviata finché la linea della cache non sia pronta a essere sostituita dall'algoritmo LRU. La scelta in questo caso è difficile e nessuna opzione è nettamente migliore delle altre. L'aggiornamento immediato nella memoria centrale dell'elemento della cache è chiamato **write-through** ("scrivi attraverso"). Questo approccio è generalmente il più semplice e il più affidabile, dato che il contenuto della memoria rimane sempre aggiornato; ciò è utile, per esempio, se si verifica un errore ed è necessario recuperare lo stato della memoria. Purtroppo però questa tecnica richiede molto traffico verso la memoria; per questo motivo si tende a utilizzare un'implementazione più sofisticata, conosciuta con il nome di **write-deferred** ("scrittura ritardata") o **write-back**.

Sempre al riguardo delle operazioni di scrittura occorre affrontare un ulteriore problema: che cosa fare se si verifica una scrittura in una locazione che in quel momento non è memorizzata nella cache? Bisogna portare il dato nella cache o basta scriverlo in memoria? Anche in questo caso nessuna delle due risposte è migliore dell'altra. La maggior parte dei progetti che ritardano la scrittura in memoria tende a portare i dati nella cache quando si verifica un fallimento in scrittura; questa tecnica è conosciuta con il nome di **write allocation** ("allocazione in scrittura"). Al contrario, quando si impiega la scrittura diretta, si tende generalmente a non allocare un elemento della cache a ogni operazione di scrittura, dato che ciò complicherebbe un'organizzazione altrimenti semplice. L'allocazione in scrittura risulta vincente solo nel caso di ripetute scritture della stessa parola o di diverse parole facenti però parte di una stessa linea della cache.

Le prestazioni della cache influenzano largamente le prestazioni generali del sistema, dato che la differenza tra la velocità della CPU e quella della memoria è molto grande. Per questa ragione la ricerca di migliori strategie per utilizzare le cache è un campo di studio ancora molto attivo (Alameldeen e Wood, 2004; Huh et al., 2004; Min et al., 2004; Nesbit e Smith, 2004; e Suh et al., 2004).

4.5.2 Predizione dei salti

I calcolatori moderni sono profondamente organizzati a pipeline. La pipeline della Figura 4.35 ha sette stadi, mentre quelle dei calcolatori ad alte prestazioni ne hanno spesso 10 o più. L'uso delle pipeline funziona in modo ottimale su codice lineare, dato che l'unità di prelievo può semplicemente leggere dalla memoria parole consecutive e spedirle all'unità di decodifica prima ancora che siano effettivamente richieste.

L'unico, piccolo, problema di questo magnifico modello è che non è, neanche in piccola parte, realistico. I programmi non sono sequenze lineari di codice, ma sono pieni d'istruzioni di salto. Consideriamo le semplici istruzioni della Figura 4.40(a). La variabile *i* viene confrontata con 0 (probabilmente il test più comune) e, a seconda del risultato, si assegna un valore diverso alla variabile *k*.

La Figura 4.40(b) mostra una traduzione in linguaggio assemblativo che verrà esaminato più avanti nel corso del libro; per ora i dettagli non sono importanti, in ogni caso a seconda della macchina e del compilatore con ogni probabilità il codice assomiglierà più o meno a quello mostrato nella Figura 4.40(b). La prima istruzione confronta *i* con 0. La seconda effettua una diramazione verso l'etichetta *Else* (l'inizio della clausola `else`) nel caso in cui *i* sia diverso da 0. La terza istruzione assegna a *k* il valore 1. La quarta effettua una diramazione verso il codice dell'istruzione successiva. Il compilatore ha opportunamente inserito in quel punto un'etichetta, *Next*, in modo che ci sia una destinazione verso la quale effettuare il salto. La quinta istruzione assegna a *k* il valore 2.

In questo esempio l'aspetto che merita una particolare osservazione è che due delle cinque istruzioni sono salti. Inoltre una di queste, `BNE`, è un salto condizionale, ovvero un salto effettuato se e soltanto se è soddisfatta una particolare condizione: in questo caso, il fatto che i due operandi della precedente istruzione, `CMP`, siano diversi. La più lunga sequenza lineare di codice è composta da due sole istruzioni. Per questi motivi risulta particolarmente difficile prelevare le istruzioni a una velocità elevata in modo da alimentare la pipeline.

```
if (i == 0)                          CMP i,0     ; confronta i con 0
    k = 1;                           BNE Else    ; salta a Else se diversi
else                        Then:    MOV k,1     ; sposta 1 in k
    k = 2;                           BR  Next    ; salta a Next
                                Else: MOV k,2     ; sposta 2 in k
                                Next:
        (a)                                (b)
```

Figura 4.40 (a) Frammento di programma. (b) Il frammento tradotto in un linguaggio assemblativo.

A prima vista potrebbe sembrare che i salti incondizionati, come l'istruzione `BR Next` della Figura 4.40(b), non costituiscano un problema. Dopo tutto non c'è alcuna ambiguità sulla destinazione da raggiungere. Perché l'unità di prelievo non può semplicemente continuare a leggere le istruzioni a partire dall'indirizzo destinazione (il punto verso il quale viene effettuato il salto)?

Il problema risiede nella natura stessa della pipeline. Nella Figura 4.35 possiamo vedere per esempio che la decodifica dell'istruzione si svolge nel secondo stadio. L'unità di prelievo deve quindi decidere dove prelevare le istruzioni successive ancor prima di conoscerne il tipo. Può scoprire di aver prelevato un salto incondizionato soltanto dopo un ciclo, nel momento in cui ha già cominciato a prelevare l'istruzione che segue il salto incondizionato. La conseguenza è che un buon numero di macchine organizzate a pipeline (come l'UltraSPARC III) ha la proprietà di eseguire l'istruzione *che segue* un salto incondizionato, anche se logicamente non dovrebbe farlo. La posizione immediatamente successiva a un salto è chiamata **posizione di ritardo**. Il Pentium II (e la macchina usata nella Figura 4.40(b)) non hanno questa proprietà, ma la complessità interna che permette di evitare questo problema è enorme. Un compilatore ottimizzato cercherà di inserire nelle posizioni di ritardo alcune istruzioni utili; spesso però non ci sono istruzioni disponibili e in tal caso il compilatore è obbligato a inserire in quel punto un'istruzione `NOP`. L'aggiunta di queste istruzioni permette di mantenere corretto il programma, rendendolo però più lento e di dimensioni maggiori.

Se i salti incondizionati provocano delle noie, quelli condizionali ne provocano di più. Non solo occorre considerare anche in questo caso delle posizioni di ritardo, ma oltretutto l'unità di prelievo viene a conoscenza dell'indirizzo in cui deve leggere in un momento della pipeline ancora più ritardato. Le prime macchine a pipeline rimanevano semplicemente in **stallo** finché non venivano a sapere se occorresse o meno effettuare la diramazione. Nel caso in cui i salti condizionali costituiscano il 20% delle istruzioni il fatto di rimanere in stallo per tre o quattro cicli a ogni salto condizionale è letale per le prestazioni.

Per queste ragioni quello che fa la maggior parte delle macchine quando incontra un salto condizionale è predire se la diramazione verrà o meno effettuata. Sarebbe bello se si potesse collegare una sfera di cristallo a uno slot PCI libero in modo da aiutare la previsione; finora però nessuno è riuscito a implementare questo approccio.

In mancanza di una simile periferica sono stati escogitati vari metodi per effettuare la predizione. Un metodo molto semplice è il seguente: si assume che debbano essere effettuati tutti i salti condizionali all'indietro, e non tutti quelli in avanti. Il ragionamento alla base della prima parte di questa strategia è che i salti all'indietro sono spesso collocati alla fine di un ciclo. La maggior parte dei cicli viene eseguita più volte e quindi, in genere, ha senso scommettere che occorra compiere un salto all'indietro per tornare all'inizio del ciclo.

La seconda parte del ragionamento si fonda su una base meno solida. Alcuni salti in avanti si verificano quando si rilevano nel software delle condizioni di errore (per esempio, non è possibile aprire un file). Gli errori sono rari, quindi la maggior parte dei salti loro associata non dovrebbe essere effettuata. Ovviamente esiste un gran numero di salti che non sono legati alla gestione degli errori e la frequenza di successi non è altrettanto buona quanto quella dei salti all'indietro. Questa regola, pur non essendo l'ideale, comunque è... meglio di niente.

Se si predice correttamente una diramazione non occorre compiere alcuna operazione particolare. L'esecuzione continua semplicemente a partire dall'indirizzo di destinazione. Il problema sorge quando la predizione si rivela errata. Determinare dove andare, e andarci, non è difficile. La parte complicata è annullare le istruzioni già eseguite che non avrebbero dovuto esserlo.

Esistono due modi di affrontare questa situazione. Il primo consiste nel consentire, dopo aver predetto un salto, l'esecuzione delle istruzioni solo finché queste non si apprestino a modificare lo stato della macchina (per esempio memorizzando un valore in un registro). Invece di sovrascrivere il registro, il valore calcolato viene inserito in un registro temporaneo (nascosto) e copiato in quello reale solo dopo aver verificato che la predizione era corretta. Il secondo metodo consiste nel registrare (per esempio in un registro temporaneo nascosto) il valore di ogni registro in procinto di essere sovrascritto, in modo che la macchina possa essere riportata allo stato in cui era nel momento in cui è stata effettuata la predizione errata. Entrambe le soluzioni sono complesse e per funzionare correttamente devono tener traccia di una quantità industriale d'informazioni. La situazione può diventare veramente complessa se si incontra un secondo salto condizionale ancor prima di scoprire se il primo era stato predetto correttamente o meno.

Predizione dinamica dei salti

Il fatto che le predizioni siano accurate è ovviamente molto importante, dato che in tal caso la CPU ha la possibilità di procedere alla massima velocità. Per questo motivo gran parte della ricerca attuale punta a migliorare gli algoritmi di predizione dei salti (Chen et al., 2003; Falcon et al., 2004; Jimenez, 2003; Parikh et al., 2004). Uno dei possibili approcci prevede che la CPU abbia (in appositi componenti hardware) una tabella della storia dei salti, nella quale tiene traccia dei salti condizionali incontrati in modo da poterli rianalizzare quando si verificano nuovamente. La versione più semplice di questo approccio è mostrata nella Figura 4.41(a). In questo caso la tabella della storia dei salti contiene un elemento per ogni istruzione di salto condizionale; l'elemento contiene l'indirizzo dell'istruzione di salto oltre a un bit che indica se è stata effettuata la diramazione l'ultima volta che l'istruzione è stata eseguita. Usando questo schema la predizione consiste semplicemente nell'assumere che il salto si comporterà allo stesso modo dell'occorrenza precedente. Se la predizione si rivela sbagliata, allora si modifica il bit nella tabella della storia dei salti.

Esistono vari modi per organizzare questa tabella, e corrispondono in sostanza alle possibili organizzazioni delle cache. Consideriamo una macchina con istruzioni a 32 bit, allineate in corrispondenza delle parole, in modo che i 2 bit meno significativi degli indirizzi di memoria siano sempre 00. Con una tabella della storia a corrispondenza diretta che contiene 2^n elementi è possibile estrarre gli $n + 2$ bit meno significativi di un'istruzione di salto e traslarli a destra di 2 bit; questo numero a n bit può essere utilizzato come indice della ta-

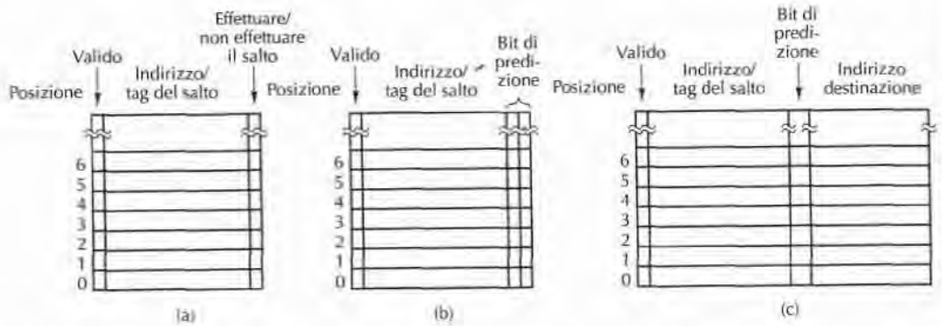


Figura 4.41 (a) Storia dei salti a 1 bit. (b) Storia dei salti a 2 bit. (c) Corrispondenza tra gli indirizzi delle istruzioni di salto e gli indirizzi destinazione.

bella della storia. L'indice così ottenuto permette di controllare se l'indirizzo memorizzato corrisponde a quello del salto. Come per le cache, anche in questo caso non è necessario memorizzare gli $n + 2$ bit meno significativi, che possono quindi essere omessi (e vengono memorizzati solamente i bit più significativi, cioè il tag). Se l'elemento cercato è presente nella tabella, la predizione utilizza il bit come previsione del futuro comportamento del salto. Se invece è presente un tag errato oppure l'elemento della tabella non è valido, si verifica un fallimento, allo stesso modo in cui avviene nelle cache. In questo caso è possibile utilizzare la regola dei salti in avanti/all'indietro.

Se la tabella della storia ha, per esempio, 4096 elementi, i salti che si trovano agli indirizzi 0, 16384, 32768, ... saranno in conflitto, in modo analogo al problema già visto con le cache. È possibile adottare la stessa soluzione: un elemento associativo a 2, a 4 o a n-vie. Come per la cache il fattore limitante è costituito da un singolo insieme a n-vie, in cui occorre confrontare un indirizzo con tutti i tag.

Se la larghezza della tabella è abbastanza grande e se vi è sufficiente associatività, allora questo schema funziona bene nella maggior parte delle situazioni. Ciononostante esiste un problema che può verificarsi in modo sistematico.

Quando si termina l'esecuzione di un ciclo, il salto che si trova alla fine verrà predetto in modo errato; ancor peggio, questo errore di previsione cambierà il bit nella tabella dei salti per indicare che in futuro occorrerà prevedere "di non effettuare il salto". Quando si rientrerà nel ciclo, alla fine della prima iterazione il salto che si trova in fondo verrà predetto in modo errato. Questo problema si può verificare spesso se il ciclo è innestato all'interno di un altro ciclo oppure nel caso in cui faccia parte di una procedura richiamata frequentemente.

Per eliminare questa predizione sbagliata possiamo dare una seconda chance all'elemento della tabella, ovvero modificare la predizione soltanto dopo che due previsioni consecutive si sono rivelate errate. Questo approccio, mostrato nella Figura 4.41(b), richiede che nella tabella della storia siano presenti due bit di predizione, uno per ciò che "si suppone" occorra fare con il salto e uno per tener traccia di ciò che si è verificato l'ultima volta.

Un modo leggermente diverso per interpretare questo algoritmo consiste nel vederlo come un automa a stati finiti con quattro stati (Figura 4.42). Dopo una serie di corrette pre-

Figura 4.42 Automa a stati finiti a 2 bit per la predizione delle diramazioni.

dizioni di "non effettuare il salto", l'automa si troverà nello stato 00 che predice di "non effettuare il salto" alla volta successiva. Se quella predizione è errata, passerà allo stato 01, ma continuerà a predire di "non effettuare il salto". Solo se anche questa predizione si rivela errata, allora la macchina passerà allo stato 11 suggerendo di effettuare la diramazione. In sostanza il bit a sinistra nello stato rappresenta la predizione, mentre il bit a destra tiene traccia dell'ultimo salto. Questa strategia prevede di utilizzare soltanto 2 bit per memorizzare la storia dei salti, ma se ne possono usare anche 4 o 8.

Già nella Figura 4.28 avevamo visto un automa a stati finiti. In effetti tutti i nostri microprogrammi possono essere descritti da questo modello in cui ogni linea rappresenta un particolare stato nel quale si può trovare l'automa e in cui è possibile compiere delle transizioni verso un insieme finito di stati. Gli automi a stati finiti sono utilizzati diffusamente in varie fasi della progettazione hardware.

Finora abbiamo assunto che la destinazione di un salto condizionale fosse nota, generalmente come indirizzo esplicito al quale andare (contenuto all'interno dell'istruzione stessa) oppure come uno spiazzamento relativo all'istruzione corrente (cioè, un valore con segno da aggiungere al contatore d'istruzioni). Questa assunzione è valida nella maggior parte dei casi, tuttavia esistono alcune istruzioni di salti condizionali che calcolano l'indirizzo di destinazione eseguendo delle operazioni aritmetiche sui registri. L'automa della Figura 4.42 predice in modo accurato se intraprendere oppure no i salti, ma queste previsioni sono inutili se l'indirizzo di destinazione è sconosciuto. Un modo per gestire questa situazione consiste nel memorizzare nella tabella della storia l'effettivo indirizzo verso il quale è stato effettuato il salto l'ultima volta, come mostra la Figura 4.41(c). Seguendo questo metodo, se la tabella indica che l'ultima volta il salto all'indirizzo 516 è stato effettuato e ha portato all'indirizzo 4000, allora, nel caso in cui la predizione dica di effettuare la diramazione, l'esecuzione continuerà nuovamente a partire dall'indirizzo 4000.

Un approccio differente per la predizione dei salti consiste nel tener traccia del fatto che gli ultimi k salti condizionali siano stati oppure no effettuati, indipendentemente dalle istruzioni alle quali erano associati. Questo numero a k bit, memorizzato in un **registro a scorrimento della storia dei salti**, viene successivamente confrontato in parallelo con tutte le chiavi a k bit degli elementi di una tabella della storia; se il confronto ha esito positivo si

utilizza la predizione trovata nella tabella. Anche se può sembrare strano questa tecnica funziona piuttosto bene.

Predizione statica delle diramazioni

Tutte le tecniche per la predizione delle diramazioni viste finora sono di tipo dinamico, cioè vengono effettuate durante l'esecuzione del programma. Uno dei vantaggi delle tecniche dinamiche è che si adattano al comportamento corrente del programma. Lo svantaggio è che richiedono un costoso hardware specializzato e una grande complessità del chip.

Una strategia alternativa consiste nel richiedere l'aiuto del compilatore. Quando il compilatore incontra un'istruzione come

```
for (i = 0; i < 1000000; i++) { ... }
```

sa con certezza che la diramazione alla fine del ciclo sarà effettuata quasi tutte le volte. Se esistesse un modo per comunicarlo all'hardware, si potrebbe risparmiare una gran quantità di lavoro.

Anche se ciò rappresenta una modifica architetturale (e non soltanto un problema d'implementazione), alcune macchine, come l'UltraSPARC III, hanno un secondo insieme d'istruzioni di salto condizionale in aggiunta a quelle ordinarie (necessarie per garantire la retrocompatibilità). Le nuove istruzioni contengono un bit attraverso cui il compilatore può specificare se ritiene che la diramazione verrà (oppure no) effettuata. Quando si incontra una di queste istruzioni l'unità di fetch esegue semplicemente l'azione specificata dal bit. Per di più queste istruzioni non richiedono spazio prezioso nella tabella delle diramazioni, riducendo di conseguenza i conflitti al suo interno.

L'ultima tecnica che consideriamo per effettuare la predizione dei salti si basa sul *profiling* (Fisher e Freudenberger, 1992). Anche questa tecnica è di tipo statico, ma, invece di delegare al compilatore il compito di prevedere quali diramazioni verranno oppure no effettuate, si manda in esecuzione il programma (di solito su un simulatore) per catturare il comportamento delle diramazioni. Questa informazione viene fornita al compilatore, che la utilizza per impostare le istruzioni speciali di salto condizionale mediante le quali può specificare all'hardware in che modo deve comportarsi.

4.5.3 Esecuzione fuori sequenza e rinomina dei registri

La maggior parte delle CPU moderne funziona sia a pipeline sia in modo superscalare, com'è illustrato nella Figura 2.6. In generale ciò comporta la presenza di un'unità di fetch che preleva istruzioni dalla memoria prima che siano necessarie, in modo da alimentare un'unità di decodifica. L'unità di decodifica attribuisce l'istruzione decodificata all'unità funzionale appropriata perché sia eseguita. In alcuni casi l'unità di decodifica può scomporre le singole istruzioni in micro-operazioni prima di distribuirle alle unità funzionali, secondo le capacità delle unità stesse.

Ovviamente l'architettura più semplice prevede che tutte le istruzioni siano eseguite nell'ordine in cui sono prelevate dalla memoria (ipotizzando per il momento che l'algoritmo di predizione delle diramazioni non sbagli mai). Tuttavia l'esecuzione in ordine non sempre fornisce prestazioni ottimali a causa delle dipendenze tra istruzioni. Le istruzioni che richiedono un valore calcolato da un'istruzione precedente non possono essere eseguite fin-

tanto che quell'istruzione non abbia prodotto il valore desiderato. In tali situazioni (dipendenze RAW) le istruzioni devono aspettare. Esistono però anche altri tipi di dipendenze, come mostreremo in seguito.

Nel tentativo di aggirare questi problemi e ottenere prestazioni migliori, alcune CPU permettono di saltare le istruzioni dipendenti per raggiungere le istruzioni successive che non lo sono. Naturalmente, l'algoritmo interno di scheduling delle istruzioni impiegato deve garantire che l'esecuzione produca lo stesso risultato che produrrebbe l'esecuzione ordinata. Vediamo come funziona il riordinamento delle istruzioni su di un esempio dettagliato.

Per illustrare la natura del problema, cominciamo con una macchina che lancia le istruzioni sempre secondo l'ordine in cui appaiono nel programma e che, inoltre, richiede vengano eseguite nello stesso ordine. L'importanza di quest'ultima caratteristica risulterà evidente in seguito.

La macchina dell'esempio ha otto registri visibili al programmatore, con nomi che vanno da `R0` a `R7`. Tutte le istruzioni aritmetiche usano tre registri: due per gli operandi e uno per il risultato, così come avviene nel Mic-4. Ipotizziamo che, se un'istruzione è decodificata nel ciclo n, allora la sua esecuzione comincia al ciclo $n + 1$. Per un'istruzione semplice, come l'addizione o la sottrazione, la scrittura del risultato nel registro di destinazione avviene alla fine del ciclo $n + 2$. Per un'istruzione più complessa, come la moltiplicazione, la scrittura avviene alla fine del ciclo $n + 3$. Per rendere l'esempio realistico, consentiamo all'unità di decodifica di lanciare fino a due istruzioni per ciclo di clock. Le CPU superscalari in commercio possono lanciare comunemente dalle quattro alle sei istruzioni per ciclo di clock.

La sequenza di esecuzione dell'esempio è mostrata nella Figura 4.43, dove la prima colonna specifica il ciclo e la seconda il numero dell'istruzione. La terza colonna elenca le istruzioni decodificate, la quarta specifica le istruzioni che vengono lanciate (con un massimo di due per ciclo di clock). L'ultima colonna contiene l'istruzione che è stata ritirata (completata) durante il ciclo corrispondente. Si tenga presente che in questo esempio si richiede che sia il lancio sia l'esecuzione avvengano in ordine, cioè l'istruzione $k + 1$ non può essere lanciata prima che sia stata lanciata l'istruzione k, e l'istruzione $k + 1$ non può essere completata (e la scrittura del risultato nel registro apposito non può avvenire) finché non viene completata l'istruzione k. Le altre sedici colonne saranno presentate tra breve.

Dopo aver decodificato un'istruzione, l'unità di decodifica deve decidere se questa può essere lanciata immediatamente o meno. Per prendere la decisione, l'unità di decodifica deve conoscere lo stato di tutti i registri. Se, per esempio, l'istruzione corrente richiede un registro il cui valore non è stato ancora computato, l'istruzione corrente non può essere lanciata e la CPU è costretta ad attendere.

Manteniamo traccia dell'uso dei registri usando uno strumento chiamato **scoreboard** ("tabella dei punteggi"), utilizzato per la prima volta nel CDC 6600. Questa tabella è provvista di un piccolo contatore per ciascun registro che rappresenta il numero d'istruzioni attualmente in esecuzione che richiedono quel particolare registro come sorgente di un operando. Se, poniamo, sono ammesse al massimo 15 istruzioni in esecuzione simultanea, allora sarà sufficiente un contatore di 4 bit. Quando un'istruzione è lanciata si incrementano le voci dello scoreboard corrispondenti ai registri dei suoi operandi, e quando un'istruzione è ritirata, le voci vengono decrementate.

Lo scoreboard contiene anche contatori per tener traccia dei registri usati come destinazioni. Poiché è permessa una sola scrittura alla volta, tali contatori possono essere

					Registri letti								Registri scritti							
C	I	Decodificata	L	R	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
1	1 2	R3=R0*R1 R4=R0+R2	1 2		1 2	1 1	 1									1 1	 1			
2	3 4	R5=R0+R1 R6=R1+R4	3 –		3 3	2 2	1 1									1 1	1 1	1 1		
3					3	2	1									1	1	1		
4				1 2 3	2 1	1 1	1										1	1 1		
5	5	R7=R1*R2	4 5			1 2	1	1	1										1 1	1
6	6	R1=R0–R2	–			2	1		1										1	1
7				4		1	1													1
8				5																
9	7	R3=R3*R1	6 –		1 1		1 1							1 1						
10					1		1							1						
11				6																
12	8	R1=R4+R4	7 –			1 1		1 1								1 1				
13						1		1								1				
14						1		1								1				
15				7																
16			8						2					1						
17									2					1						
18				8																

Figura 4.43 CPU superscalare che lancia e ritira le istruzioni in ordine (C = ciclo; L = lanciata; R = ritirata).

lunghi un solo bit. Le ultime sedici colonne sulla destra della Figura 4.43 mostrano lo scoreboard.

Nelle macchine reali lo scoreboard registra anche l'uso delle unità funzionali, per evitare il lancio di un'istruzione per la quale non vi siano unità funzionali disponibili. Per semplicità, qui consideriamo il caso in cui ci siano sempre unità funzionali appropriate disponibili, così possiamo omettere le unità funzionali dallo scoreboard.

La prima riga della Figura 4.43 mostra I1 (istruzione 1), che moltiplica `R0` e `R1` e pone il risultato in `R3`. Dal momento che nessuno di questi registri è già in uso, l'istruzione viene lanciata e lo scoreboard è aggiornato con l'annotazione che `R0` e `R1` sono usati in lettura e `R3` in scrittura. Nessuna delle istruzioni successive può scrivere in alcuno di questi registri o leggere il contenuto di `R3` finché I1 non venga ritirata. Trattandosi di una moltiplicazione, l'istruzione verrà completata al termine del ciclo 4. I valori delle righe dello scoreboard

riflettono lo stato dei registri al lancio delle diverse istruzioni. Le caselle vuote rappresentano il valore 0 dei contatori.

Poiché la macchina dell'esempio è una macchina superscalare che può lanciare due istruzioni per ciclo, la seconda istruzione viene lanciata anch'essa durante il ciclo 1. Tale istruzione somma `R0` e `R1`, salvando il risultato in `R4`. Per stabilire se un'istruzione può essere lanciata si applicano le regole seguenti:

1. se un operando è in fase di scrittura, non lanciare (dipendenza RAW)
2. se il registro del risultato è in lettura, non lanciare (dipendenza WAR)
3. se il registro del risultato è in scrittura, non lanciare (dipendenza WAW).

Abbiamo già parlato di dipendenza RAW, che si verifica quando un'istruzione deve usare come operando il risultato di un'altra istruzione non ancora completata. Gli altri due tipi di dipendenze sono meno gravi: si tratta essenzialmente di conflitti di risorse. In una **dipendenza di tipo WAR** (*Write After Read*) un'istruzione cerca di sovrascrivere un registro che un'istruzione precedente potrebbe non aver terminato di leggere, e la **dipendenza WAW** (*Write After Write*) è simile. Entrambe possono essere evitate richiedendo che la seconda istruzione salvi il suo risultato da qualche altra parte (magari temporaneamente). Un'istruzione può essere lanciata, se non si verifica nessuna delle tre dipendenze e l'unità funzionale di cui ha bisogno è disponibile. Nel nostro caso l'istruzione 2 usa il registro `R0` che è in lettura da parte di un'istruzione in corso di svolgimento, ma questa sovrapposizione è consentita e così l'istruzione viene lanciata. Analogamente, la terza istruzione può essere lanciata durante il ciclo 2.

Veniamo ora all'istruzione 4, che richiede l'uso di `R4`. Sfortunatamente notiamo che, alla riga 3, il registro R4 è in fase di scrittura. In questo caso ci troviamo di fronte a una dipendenza RAW, perciò l'unità di decodifica va in stallo finché `R4` non torni disponibile. Durante l'attesa, l'unità smette di prelevare istruzioni dall'unità di fetch. Quando il suo buffer si riempie, l'unità smette di eseguire il prefetch.

È utile notare che l'istruzione seguente secondo l'ordine del programma, cioè la 5, non causa alcun conflitto con le istruzioni in corso di svolgimento. Se l'architettura non richiedesse il lancio delle istruzioni in ordine, sarebbe possibile decodificare e lanciare l'istruzione 5.

Osserviamo quindi che cosa succede durante il ciclo 3. L'istruzione 2 termina alla fine del ciclo 3, trattandosi di un'addizione (due cicli). Sfortunatamente però non può essere ritirata (liberando `R4` per l'uso da parte dell'istruzione 4). Perché? La ragione è che questa architettura richiede anche il ritiro ordinato delle istruzioni. Perché? Quale problema potrebbe sorgere dal salvare il risultato in `R4` e marcarlo come disponibile?

La risposta è sottile e al contempo importante. Se le istruzioni possono essere completate fuori sequenza, in caso venisse sollevato un interrupt, sarebbe molto difficile salvare lo stato della macchina per poterlo ripristinare successivamente. In particolare, non sarebbe possibile stabilire che siano state eseguite tutte le istruzioni fino a un certo indirizzo e non quelle successive. Questa capacità è detta **interrupt preciso** ed è una caratteristica auspicabile in una CPU (Moudgill e Vassiliadis, 1996). Il ritiro delle istruzioni fuori sequenza rende gli interrupt imprecisi, ed è questo il motivo per cui alcune macchine richiedono il completamento in ordine delle istruzioni.

Tornando all'esempio, alla fine del ciclo 4 le tre istruzioni in esecuzione possono essere ritirate, e l'istruzione 4 può essere infine lanciata durante il ciclo 5, insieme alla 5 ap-

pena decodificata. Ogniqualvolta un'istruzione viene ritirata, l'unità di decodifica deve controllare se c'è un'istruzione in stallo che può essere lanciata.

Durante il ciclo 6 l'istruzione 6 è in stallo perché deve scrivere nel registro R1 che è occupato, e la sua esecuzione comincia solo al ciclo 9. Il completamento dell'intera sequenza delle otto istruzioni richiede diciotto cicli a causa di molte dipendenze, nonostante l'hardware sia capace di lanciare due istruzioni a ogni ciclo. D'altra parte, si noti che scorrendo la colonna *L* della Figura 4.43 si evince come tutte le istruzioni siano state lanciate in ordine; la colonna *R* mostra come siano tutte state ritirate in ordine.

Prendiamo ora in esame un'altra architettura che ammette l'esecuzione fuori sequenza. Ora le istruzioni possono essere sia lanciate sia ritirate fuori sequenza. La stessa sequenza di otto istruzioni è mostrata nella Figura 4.44, laddove sono ora consentiti il lancio e il ritiro fuori sequenza.

La prima differenza si verifica al ciclo 3. Anche se l'istruzione 4 è in stallo, è permesso decodificare e lanciare la 5 visto che non è in conflitto con nessuna istruzione in esecuzione. A ogni modo, la possibilità di saltare alcune istruzioni causa un nuovo problema.

Si immagini che l'istruzione 5 utilizzi un operando calcolato dall'istruzione 4, l'istruzione saltata; con l'attuale scoreboard non lo avremmo notato. Bisogna quindi estendere la tabella per tenere traccia dei salvataggi effettuati dalle istruzioni saltate. A tal fine è possibile aggiungere un bit per registro (non mostrati in figura), per annotare i salvataggi effettuati dalle istruzioni in stallo. La regola per il lancio delle istruzioni deve ora essere estesa per

					Registri letti								Registri scritti							
C	I	Decodificata	L	R	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
1	1 2	R3=R0*R1 R4=R0+R2	1 2		1 2	1 1	 1									1 1	 1			
2	3 4	R5=R0+R1 R6=R1+R4	3 –		3 3	2 2	1 1									1 1	1 1	1 1		
3	5 6	R7=R1*R2 S1=R0–R2	5 6	 2	3 4 3	3 3 3	2 3 2									1 1 1	1 1	1 1 1		1 1 1
4	 7 8	 R3=R3*S1 S2=R4+R4	4 – 8	 1 3	3 3 3 2 1	4 4 4 3 2	2 2 2 2 2		1 1 3 3 3							1 1 1		1 1 1 1	1 1 1 1 1	1 1 1 1 1
5				6		2	1		3					1					1	1
6			7	 4 5 8		2 1	1 1	1 1 1 1	3 2 2					1 1 1		1 1 1 1			1	1 1
7								1								1				
8								1								1				
9				7																

Figura 4.44 Operazioni di una CPU superscalare che lancia e ritira le istruzioni fuori sequenza (C = ciclo; L = lanciata; R = ritirata).

prevenire il lancio delle istruzioni i cui operandi sono il risultato di un'istruzione precedente che è stata saltata.

Tornando alle istruzioni 6, 7 e 8 nella Figura 4.43, notiamo che la 6 restituisce un valore in R1 che è richiesto dall'istruzione 7. Notiamo anche che tale valore non è mai più richiesto perché l'istruzione 8 sovrascrive R1. Non c'è motivo di usare R1 per contenere il risultato dell'istruzione 6. Per di più R1 è una pessima scelta come registro intermedio, nonostante sia ragionevole per un compilatore o per un programmatore abituato all'idea di esecuzione sequenziale senza sovrapposizione d'istruzioni.

Nella Figura 4.44 introduciamo una nuova tecnica per risolvere questo problema: la **rinomina dei registri**. Una saggia unità di decodifica preferirà all'uso di R1 nell'istruzione 6 (ciclo 3) e 7 (ciclo 4) l'uso di un registro segreto, S1, invisibile al programmatore. In questo modo l'istruzione 6 può essere lanciata in modo concorrente con la 5. Le CPU moderne hanno sovente dozzine di registri segreti destinati alla rinomina di registri. Questa tecnica riesce in genere a eliminare le dipendenze WAR e WAW.

In corrispondenza dell'istruzione 8 usiamo ancora la rinomina dei registri. Questa volta R1 è rinominato come S2 di modo che l'addizione possa cominciare prima che R1 sia libero, alla fine del ciclo 6. Nel caso in cui R1 fosse a questo punto la vera destinazione del risultato, il contenuto di S2 potrebbe sempre essere ricopiato in R1 in tempo utile. Oppure, soluzione ancora migliore, è possibile rinominare le sorgenti di tutte le istruzioni future che avranno bisogno di R1 alla locazione dove il suo valore è effettivamente memorizzato. In ogni caso, con questa tecnica l'addizione dell'istruzione 8 può essere lanciata più prontamente.

Su molte macchine reali il meccanismo di rinomina è profondamente connaturato con il modo in cui sono organizzati i registri: ci sono molti registri segreti e una tabella che mette in corrispondenza i registri visibili al programmatore con quelli segreti. Così il registro usato effettivamente al posto di R0 viene individuato, per esempio, guardando alla voce di posto 0 nella tabella delle corrispondenze. In questo modo non c'è un vero registro R0, ma solo un legame tra il nome R0 e uno dei registri segreti. Al fine di evitare le dipendenze tale legame cambia frequentemente nel corso dell'esecuzione.

Dalla quarta colonna della Figura 4.44 si evince che le istruzioni non sono state lanciate né ritirate in ordine. La conclusione di questo esempio è semplice: attraverso l'utilizzo dell'esecuzione fuori sequenza e della rinomina di registri è stato possibile abbreviare il calcolo di un fattore due.

4.5.4 Esecuzione speculativa

Nelle sezioni precedenti abbiamo introdotto il concetto di riordino delle istruzioni, allo scopo di migliorare le prestazioni. Sebbene non sia stato dichiarato in modo esplicito, finora l'attenzione è stata rivolta al riordinamento che avviene all'interno di un singolo **blocco elementare**. Passiamo adesso a un'analisi più approfondita di questo aspetto.

I programmi sono suddivisibili in blocchi elementari, ciascuno dei quali è costituito da una sequenza lineare d'istruzioni con un punto d'ingresso all'inizio e un punto di uscita alla fine. Un blocco elementare non contiene al suo interno alcuna struttura di controllo (come i costrutti `if` o `while`) di modo che la sua traduzione in linguaggio macchina non presenti nessuna diramazione. Il collegamento fra diversi blocchi elementari è costituito dalle strutture di controllo.

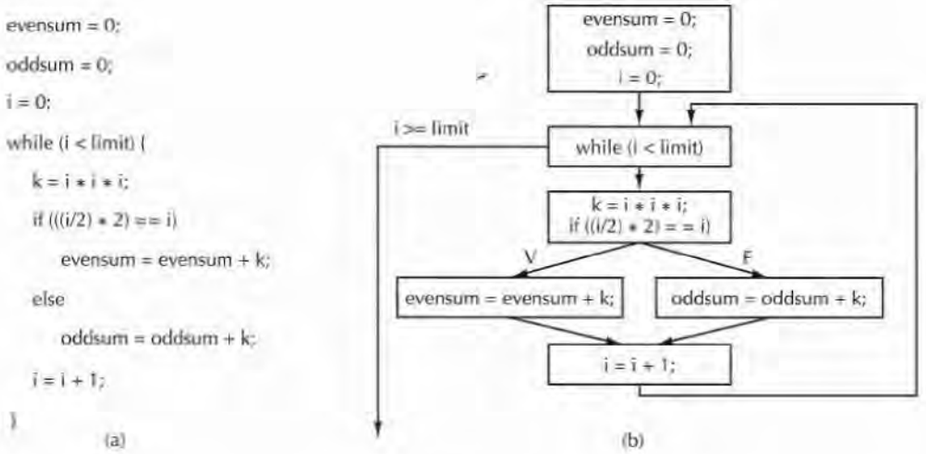


Figura 4.45 (a) Frammento di programma. (b) Grafico a blocchi corrispondente.

Un programma in questa forma può essere rappresentato attraverso un grafo orientato (Figura 4.45). In questo esempio le somme dei numeri pari e dei numeri dispari vengono calcolate separatamente fino a un certo limite e memorizzate rispettivamente in *evensum* e *oddsum*. All'interno di ciascun blocco elementare il riordino delle istruzioni descritto nella sezione precedente funziona in maniera corretta.

Il problema è che la maggior parte dei blocchi elementari è di breve lunghezza e quindi non vi è al loro interno sufficiente parallelismo da poter sfruttare in modo efficace. Di conseguenza, il passo successivo per cercare di utilizzare tutti i cicli disponibili consiste nel permettere al riordinamento di superare i limiti che separano i blocchi elementari. Il guadagno maggiore si ottiene quando un'istruzione potenzialmente lenta può essere spostata più in alto nel grafo di modo che la sua esecuzione possa iniziare in modo anticipato. Questa potrebbe essere un'istruzione `LOAD`, un'operazione in virgola mobile, oppure l'inizio di una lunga catena di dipendenze. La tecnica di anticipare l'esecuzione del codice prima di una diramazione è nota come **slittamento**.

Immaginiamo che nella Figura 4.45 tutte le variabili siano memorizzate nei registri tranne *evensum* e *oddsum* (per mancanza di registri). Potrebbe aver senso spostare le loro istruzioni di `LOAD` all'inizio del ciclo, prima ancora di calcolare *k*, per farle cominciare anticipatamente di modo che i loro valori siano già disponibili nel momento in cui vengano richiesti. Ovviamente, durante ogni iterazione, solo uno di loro sarà effettivamente necessario, mentre l'altra operazione di `LOAD` verrà sprecata; se però sia la cache sia la memoria sono strutturate a pipeline e se vi sono cicli disponibili, potrebbe valerne la pena. L'esecuzione di parti del codice prima ancora di sapere se saranno effettivamente richieste è chiamata **esecuzione speculativa**. Per poter usare questa tecnica sono necessari sia il supporto da parte del compilatore e dell'hardware sia alcune estensioni dell'architettura. È compito del compilatore spostare esplicitamente le istruzioni, dato che un loro riordinamento che scavalchi i limiti dei blocchi elementari è generalmente al di là delle possibilità dell'hardware.

L'esecuzione speculativa introduce alcuni problemi interessanti. Fra questi, è essenziale che nessuna delle istruzioni speculative produca risultati irrevocabili, dato che in un secondo momento potrebbe risultare che non dovevano essere eseguite. Nell'esempio della Figura 4.45 è corretto effettuare il fetch di *evensum* e *oddsum*, e anche calcolare la somma non appena *k* è disponibile (prima del costrutto `if`), ma non è corretto trasferire il risultato in memoria. In sequenze d'istruzioni più complesse, un modo comune per evitare che una parte del codice speculativo scriva nei registri prima ancora di sapere se ciò è voluto, consiste nel rinominare tutti i registri di destinazione utilizzati dal codice speculativo. In questo modo vengono modificati solo i registri di lavoro, così che non sorgano problemi se in ultima istanza il codice risulta non richiesto. Se al contrario il codice è necessario, allora i registri di lavoro vengono copiati nei registri di destinazione corretti. Come si può immaginare, non è semplice utilizzare la tecnica dello scoreboard per tener traccia di tutto ciò, ma può tuttavia essere realizzato disponendo di hardware sufficiente.

In ogni caso esiste un altro problema introdotto dal codice speculativo che non può essere risolto attraverso la semplice rinomina dei registri. Che cosa succede se l'esecuzione di un'istruzione speculativa causa un'eccezione? Una situazione fastidiosa, anche se non irrimediabile, si verifica quando un'istruzione `LOAD` causa un fallimento in una cache con blocchi di grandi dimensioni (per esempio 256 byte) e di una memoria molto più lenta sia della CPU sia della cache. Se un'istruzione di `LOAD` realmente necessaria pone la macchina in attesa per vari cicli, durante la fase di caricamento del blocco della cache, ciò deve essere considerato normale, dato che la parola è stata effettivamente richiesta. Al contrario, mettere in stallo una macchina durante il fetch di una parola che successivamente non risulterà necessaria è controproducente, al punto che un numero eccessivo di queste "ottimizzazioni" potrebbe rendere la CPU più lenta di quanto non lo sarebbe se non si facesse ricorso alle stesse. Se la macchina è dotata di memoria virtuale (trattata nel Capitolo 6) allora un'istruzione di `LOAD` speculativa potrebbe addirittura causare un errore di pagina e quindi richiedere un accesso al disco per ottenere la pagina richiesta. Dato che i falsi errori di pagina possono avere un effetto devastante sulle prestazioni è importante evitare che si manifestino.

Una soluzione utilizzata in alcune macchine moderne consiste nell'avere una speciale istruzione speculativa di `LOAD`, `SPECULATIVE-LOAD`, che cerchi di prelevare la parola dalla cache, ma che non compia alcuna azione se non la trova. Se, nel momento in cui viene richiesto, il valore è presente, allora può essere utilizzato; altrimenti l'hardware deve andare a recuperarlo sul momento. In questo modo il fallimento della cache non produce alcuna penalità se alla fine il valore risulta essere non necessario.

Una situazione decisamente peggiore è rappresentata dalla seguente istruzione

```
if (x > 0) z = y/x;
```

dove *x*, *y* e *z* sono variabili in virgola mobile. Si supponga che tutte le variabili siano prelevate e memorizzate in anticipo nei registri e che venga usata la tecnica dello slittamento del codice per spostare l'esecuzione della lunga operazione in virgola mobile prima del test `if`. Sfortunatamente *x* vale 0 e l'eccezione generata dalla divisione per zero fa terminare il programma. Il risultato è che l'uso dell'esecuzione speculativa ha fatto fallire un programma corretto. La cosa peggiore è che ciò è avvenuto nonostante il programmatore avesse previsto il problema e programmato esplicitamente il codice in modo da evitarlo. Difficilmente una simile situazione può rendere felici i programmatori.

Una possibile soluzione consiste nell'avere delle versioni speciali delle istruzioni che possano sollevare delle eccezioni, e nell'aggiungere a ogni registro un bit chiamato **poison bit** ("bit avvelenato"). Quando un'istruzione speculativa speciale fallisce, essa, al posto di causare un'eccezione, assegna il valore 1 al poison bit del registro risultato. Se in un secondo momento un'istruzione regolare utilizza questo registro allora viene sollevata un'eccezione (com'è giusto che sia). In ogni caso, se il risultato non è mai utilizzato, allora il poison bit può essere cancellato senza provocare alcun danno.

4.6 Esempi del livello di microarchitettura

In questo paragrafo mostreremo alcuni brevi esempi di tre processori all'avanguardia e vedremo come vengano impiegati al loro interno i concetti analizzati nel corso di questo capitolo. Dato che le macchine reali sono enormemente complesse (contengono milioni di porte logiche) la loro descrizione sarà necessariamente limitata agli aspetti essenziali. Le macchine in esame sono le solite usate finora: il Pentium 4, l'UltraSPARC III e il processore 8051.

4.6.1 Microarchitettura della CPU Pentium 4

Dall'esterno il Pentium 4 appare come una macchina CISC tradizionale, con un insieme d'istruzioni molto esteso e poco maneggevole, con operazioni su interi a 8, 16 e 32 bit e operazioni in virgola mobile a 32 e 64 bit. I registri visibili sono solo otto e sono tutti diversi tra loro. La lunghezza delle istruzioni varia da 1 a 17 byte. In breve, si tratta di un'architettura ereditata dal passato che sembra fare ogni cosa in modo errato.

Ciononostante il Pentium 4 contiene al suo interno un nucleo RISC moderno, semplice, efficace e altamente strutturato a pipeline; la velocità di clock è estremamente elevata ed è presumibile che essa continuerà ad aumentare anche nei prossimi anni. È sorprendente come gli ingegneri della Intel siano riusciti a costruire un processore moderno per implementare un'architettura di vecchia concezione.

Descrizione della microarchitettura NetBurst

La microarchitettura del Pentium 4, chiamata **NetBurst**, rappresenta una rottura completa rispetto alla precedente, P6, utilizzata nel Pentium Pro, nel Pentium II e nel Pentium III, e costituisce la base a partire dalla quale Intel costruirà nei prossimi anni le nuove microarchitetture. La Figura 4.46 mostra una visione semplificata dell'architettura del Pentium 4 (corrispondente più o meno alla Figura 1.12).

Il Pentium 4 è costituito da quattro parti principali: il sottosistema di memoria, il *front end*, il controllo dell'esecuzione fuori sequenza e le unità esecutive. Passiamo ad analizzare questi componenti in senso antiorario a partire da quello situato in alto a sinistra nella figura.

Il sottosistema di memoria contiene la cache L2 (di secondo livello) unificata oltre alla logica necessaria per accedere, attraverso il bus della memoria, alla RAM esterna. Nella prima generazione del Pentium 4 essa era di 256 KB; nella seconda era di 512 KB; nella terza di 1 MB. Questa cache è associativa a 8 vie, con blocchi da 128 byte. Quando una richiesta alla cache L2 fallisce, partono due trasferimenti da 64 byte verso la memoria principale per

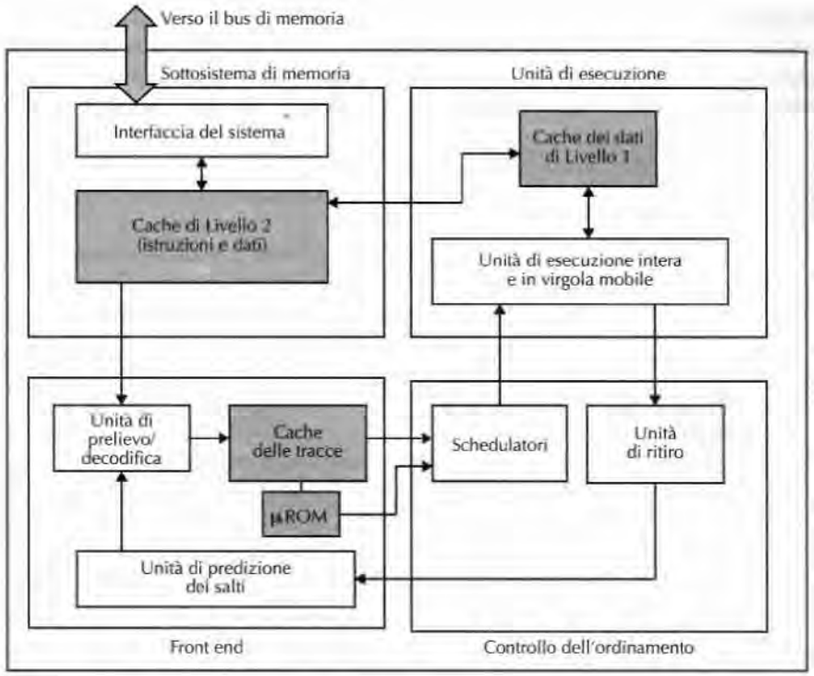


Figura 4.46 Diagramma a blocchi del Pentium 4.

prelevare i blocchi richiesti. La cache L2 è una cache di tipo write-back; ciò significa che quando un blocco è modificato, il nuovo contenuto non viene scritto in memoria finché il blocco non vada riscritto.

Alla cache è associata l'unità di prefetch (non mostrata in figura), il cui compito consiste nel prelevare i dati dalla memoria principale e copiarli nella cache L2 prima che siano richiesti. I dati possono viaggiare dalla cache L2 alle altre cache ad alte velocità. Durante ogni ciclo di clock può aver inizio un nuovo prelievo dalla cache L2, e quindi, nel caso di un processore da 3 GHz, la cache L2 può permettere un trasferimento verso le altre cache di 1,5 miliardi di blocchi di 64 byte al secondo, corrispondenti a una larghezza di banda di 96 GB/s.

Nella Figura 4.46, sotto il sottosistema di memoria, si trova il front end, il cui compito è il prelievo delle istruzioni dalla cache L2 e la loro decodifica nell'ordine del programma. Le istruzioni ISA del Pentium sono scomposte in una sequenza di micro-operazioni di tipo RISC. Per le istruzioni più semplici l'unità di prelevamento/decodifica determina internamente le micro-operazioni necessarie, mentre per istruzioni più complesse la sequenza di micro-operazioni è letta da una micro-ROM. In entrambi i modi, ogni istruzione Pentium ISA viene convertita in una sequenza di micro-operazioni che possono essere eseguite dal nucleo RISC del processore. Questo meccanismo permette di colmare il divario che esiste tra un vecchio insieme d'istruzioni di tipo CISC e un moderno percorso dati RISC.

Le micro-operazioni decodificate vengono memorizzate nella **cache delle tracce**, che è una cache d'istruzioni di primo livello. Immagazzinando nella cache la sequenza di micro-operazioni al posto dell'istruzione originaria si evita di dovere decodificare una seconda volta la stessa istruzione quando viene eseguita estraendola dalla cache delle tracce. Questo approccio rappresenta una delle differenze principali tra la microarchitettura NetBurst e il P6 (che si limitava a conservare le istruzioni del Pentium 4 nella cache delle istruzioni di primo livello). Anche la predizione delle diramazioni viene effettuata durante questa fase.

Le istruzioni sono trasferite dalla cache delle tracce allo schedulatore secondo l'ordine del programma, ma non sono necessariamente lanciate in quell'ordine. Quando lo schedulatore incontra una micro-operazione che non può essere eseguita, la trattiene, pur continuando a processare il flusso d'istruzioni per lanciare le istruzioni successive per cui sono disponibili tutte le risorse (registri, unità funzionali, ecc.). Anche la rinomina dei registri viene effettuata in questa fase per permettere alle istruzioni con dipendenza WAR o WAW di continuare senza alcun ritardo.

Sebbene le istruzioni possano essere lanciate fuori sequenza, i requisiti dell'architettura Pentium per gestire le interruzioni precise fanno sì che le istruzioni ISA debbano essere ritirate (cioè aver reso visibili i loro risultati) in ordine. L'unità di ritiro gestisce questo compito.

Nel quadrante in alto a destra della figura si trovano le unità di esecuzione che processano istruzioni su interi, in virgola mobile e specializzate. Sono presenti varie unità di esecuzione che funzionano in parallelo. Esse ottengono i loro dati dall'insieme dei registri (*register file*) e dalla cache di dati L1.

Pipeline NetBurst

Nella Figura 4.47 è possibile vedere una rappresentazione più dettagliata della microarchitettura NetBurst, in cui è rappresentata anche la pipeline. Nella parte alta dell'immagine si trova il front end, il cui compito è quello di prelevare istruzioni dalla memoria e prepararle per l'esecuzione. Il front end è alimentato dalle nuove istruzioni Pentium che provengono dalla cache L2, 64 bit alla volta, e le decodifica in micro-operazioni per memorizzarle nella cache delle tracce che ne può contenere 12K. Le prestazioni fornite da una cache di queste dimensioni sono paragonabili a quelle di una cache convenzionale L1 di dimensione compresa tra 8 e 16 KB. La cache delle tracce mantiene, all'interno di ogni linea della traccia, gruppi di sei micro-operazioni. Le micro-operazioni di una stessa linea della traccia verranno eseguite in ordine, anche se possono provenire da istruzioni ISA distanti l'una dall'altra migliaia di byte. Inoltre, se la sequenza di micro-operazioni è più lunga, è possibile collegare fra loro più linee della traccia.

Quando un'istruzione Pentium ISA richiede più di quattro micro-operazioni, nella cache delle tracce non viene memorizzata la sua decodifica, bensì un indicatore che segnala di cercare le micro-operazioni nella ROM del microcodice. In questo modo le micro-operazioni vengono inserite nell'unità logica di esecuzione fuori sequenza, utilizzando dalla cache delle tracce le istruzioni ISA precedentemente decodificate oppure cercando al momento all'interno della ROM del microcodice le istruzioni ISA più complesse, come lo spostamento di una stringa.

Se l'unità di decodifica incontra una diramazione condizionale, cerca la destinazione predetta all'interno del **BTB L1** (*Branch Target Buffer*, "buffer della destinazione dei salti")

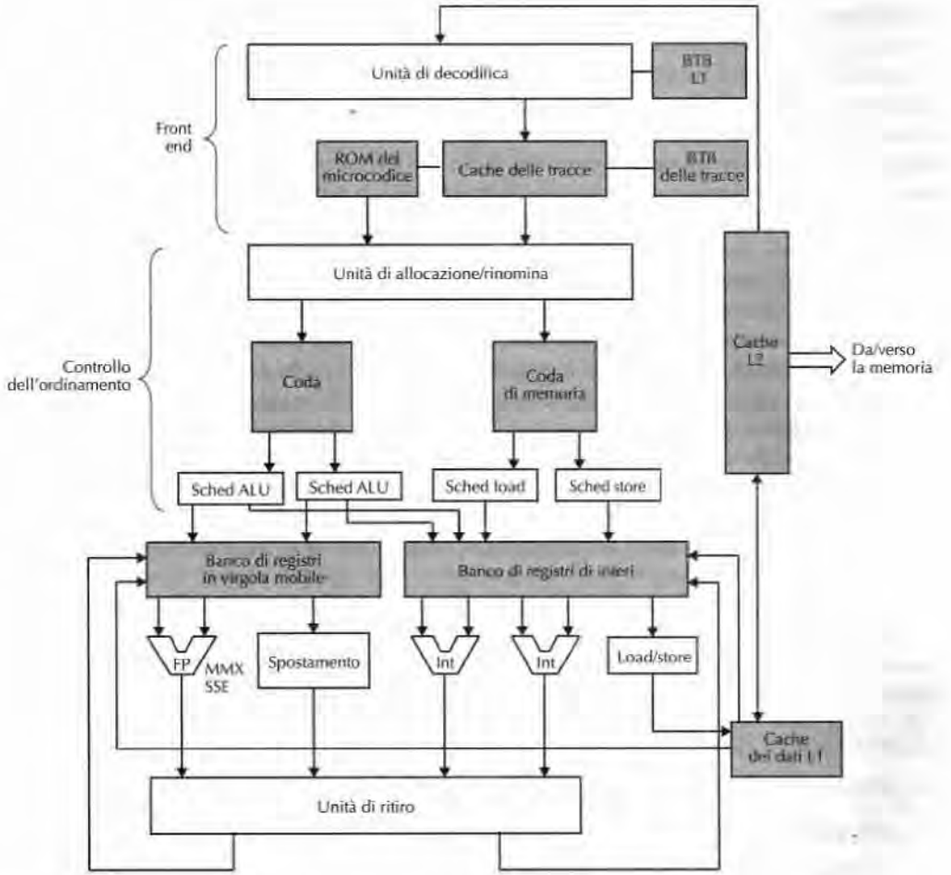


Figura 4.47 Vista semplificata del percorso dati del Pentium 4.

e continua a partire da questo indirizzo. Il BTB L1 memorizza i 4K salti più recenti. Se l'istruzione di diramazione non è presente nella tabella, si utilizza la predizione statica. Quando si incontra un salto all'indietro si assume che faccia parte di un ciclo e viene quindi effettuato; l'accuratezza di questo tipo di predizione è estremamente elevata. Quando invece si incontra un salto in avanti si assume che faccia parte di un'istruzione `if` e si sceglie di non effettuare la diramazione. In questo caso l'accuratezza della predizione statica è molto più bassa rispetto a quella dei salti all'indietro. Il **BTB delle tracce** viene utilizzato per prevedere dove porteranno le micro-operazioni di salto.

La seconda parte della pipeline, l'unità di controllo dell'esecuzione fuori sequenza, è alimentata dalla cache delle tracce, che contiene 12K micro-operazioni. Man mano che le micro-operazioni giungono dal front end, l'unità di **allocazione** e **rinomina** ne tiene traccia in una tabella, chiamata **ROB** (*ReOrder Buffer*, "buffer di riordinamento"), di 128 ele-

menti. Ognuno di questi memorizza lo stato della micro-operazione prima che essa venga ritirata. L'unità di allocazione e rinomina effettua quindi un controllo per verificare se le risorse richieste dalla micro-operazione sono disponibili. Se lo sono, la micro-operazione viene inserita in una delle due code di esecuzione: una per le micro-operazioni di memoria e una per tutte le altre. In caso contrario la micro-operazione viene ritardata, ma quelle successive vengono in ogni caso elaborate, portando dunque a un'esecuzione fuori sequenza delle micro-operazioni. L'obiettivo di questa strategia è quello di tenere il più possibile occupate le unità funzionali. In un qualsiasi momento è possibile gestire fino a 126 istruzioni, di cui fino a 48 possono essere di scrittura in memoria e 24 di lettura dalla memoria.

A volte una micro-operazione può rimanere in stallo, poiché deve scrivere in un registro che in quel momento viene letto o scritto da una micro-operazione precedente. Come abbiamo già visto questi conflitti sono chiamati dipendenze WAR e WAV. Attraverso la rinomina della sua destinazione è possibile effettuare lo scheduling della micro-operazione affinché venga immediatamente eseguita; con questa tecnica la scrittura avviene in uno dei 120 registri di lavoro invece che all'interno della destinazione prevista, che risulta essere occupata. Se non ci sono registri disponibili o se la micro-operazione ha una dipendenza RAW (che non può mai essere risolta), l'allocatore annota la natura del problema all'interno dell'elemento della tabella ROB. La micro-operazione viene poi inserita in una delle code di esecuzione nel momento in cui tutte le risorse richieste si rendono disponibili.

Quando le micro-operazioni sono pronte per essere eseguite, l'unità di allocazione e rinomina le inserisce nelle due code di esecuzione. Alle estremità opposte di queste code si trovano quattro **schedulatori** di micro-operazioni, ognuno dei quali si occupa di pianificare l'utilizzo di alcune risorse specifiche, nel modo seguente:

1. schedulatore 1: la ALU 1 e l'unità per gli spostamenti in virgola mobile
2. schedulatore 2: la ALU 2 e l'unità di esecuzione in virgola mobile
3. schedulatore 3: le istruzioni di caricamento
4. schedulatore 4: le istruzioni di memorizzazione.

Dato che gli schedulatori e le ALU funzionano al doppio della frequenza nominale del clock, i primi due schedulatori possono elaborare due micro-operazioni per ciclo di clock. Avendo due ALU intere, ciascuna delle quali lavora al doppio della velocità, un Pentium 4 a 3 GHz è in grado di eseguire 12 miliardi di operazioni su interi al secondo. È proprio la possibilità di raggiungere velocità tanto elevate che spinge il controllo dell'esecuzione fuori sequenza a cercare costantemente del lavoro da assegnare alle ALU. Le istruzioni di caricamento e memorizzazione condividono un'unica unità di esecuzione funzionante al doppio della frequenza, capace di lanciare in ogni ciclo un'istruzione di caricamento e una di memorizzazione. Nel caso migliore è dunque possibile lanciare sei micro-operazioni su interi per ciclo di clock, oltre ad alcune istruzioni su valori in virgola mobile.

Le due ALU intere non sono identiche. La ALU 1 può eseguire tutte le operazioni aritmetiche e logiche, oltre alle diramazioni. La ALU 2 può invece eseguire solamente le istruzioni di addizione, sottrazione, scorrimento e rotazione. Esistono delle differenze anche tra le due unità in virgola mobile. La prima può eseguire solamente gli spostamenti e le istruzioni SSE. La seconda può eseguire le operazioni aritmetiche in virgola mobile, le istruzioni MMX e le istruzioni SSE.

Le ALU e le unità in virgola mobile sono alimentate da una coppia di banchi di registri (*register file*) di 128 elementi; uno per i valori interi, e l'altro per quelli in virgola mobile. Questi registri forniscono tutti gli operandi per le istruzioni da eseguire e servono anche da deposito per i risultati. Al livello ISA sono visibili otto registri (`EAX`, `EBX`, `ECX`, `EDX`, e così via), ma, per via della rinomina dei registri, gli otto che contengono effettivamente i "veri" valori variano nel tempo a seconda di come cambia la corrispondenza durante l'esecuzione.

La cache dei dati L1 fa parte dei circuiti ad alta velocità (2x). È una cache da 8 KB che può contenere valori interi, in virgola mobile o qualsiasi altro tipo di dati. Diversamente dalla cache delle tracce essa non è decodificata in alcun modo, ma semplicemente mantiene una copia dei byte presenti in memoria. La cache di dati L1 è una cache associativa a 4 vie con linee da 64 byte. È una cache di tipo write-thorugh, nel senso che le linee modificate vengono copiate immediatamente nella cache L2. La cache può gestire un'operazione di lettura e una di scrittura per ciclo di clock. Quando una parola richiesta non è presente nella cache L1, viene spedita una richiesta alla cache L2, che o fornisce immediatamente il valore oppure preleva dalla memoria la linea della cache e in seguito invia la risposta. In un dato istante possono essere gestite fino a quattro richieste dalla cache L1 alla cache L2.

Dato che le micro-operazioni vengono eseguite fuori sequenza, le operazioni di scrittura nella cache L1 non sono permesse finché non siano state ritirate tutte le istruzioni che precedono quella che ha generato la scrittura. L'**unità di ritiro** ("completamento") ha il compito di ritirare in ordine le istruzioni e di tener traccia del punto in cui ci si trova. Se si verifica un interrupt le istruzioni non ancora ritirate vengono annullate; in questo modo il Pentium 4 gode della proprietà che, quando si verifica un interrupt, esiste un punto in cui tutte le istruzioni precedenti sono state completate e nessuna di quelle successive produce alcun effetto.

Se viene ritirata un'istruzione di memorizzazione mentre qualche istruzione che la precede è ancora in esecuzione, non è possibile aggiornare la cache L1. I risultati vengono quindi inseriti in un buffer per le memorizzazioni pendenti; questo buffer è composto da 24 elementi che corrispondono a 24 possibili memorizzazioni contemporaneamente in esecuzione. In seguito, se un'istruzione di caricamento prova a leggere i dati memorizzati, può esserle restituito il valore contenuto nel buffer per le memorizzazioni pendenti, anche se non è ancora stato aggiornato all'interno della cache L1. Questo processo è chiamato avanzamento **store-to-load**.

A questo punto dovrebbe essere evidente che il Pentium 4 ha una microarchitettura molto complessa, progettata per poter eseguire il vecchio insieme d'istruzioni Pentium su un moderno nucleo di tipo RISC con una profonda struttura a pipeline. Per raggiungere questo scopo suddivide le istruzioni Pentium in micro-operazioni, le memorizza all'interno di una cache e ne inserisce tre alla volta nella pipeline, per essere eseguite da un insieme di ALU; in condizioni ottimali queste unità sono in grado di eseguire fino a sei micro-operazioni per ciclo. Le micro-operazioni sono eseguite fuori sequenza, ma vengono ritirate in ordine e anche la memorizzazione dei risultati all'interno delle cache L1 e L2 avviene in ordine. In Hinton et al. (2004) è possibile trovare maggiori informazioni sulla microarchitettura NetBurst.

4.6.2 Microarchitettura della CPU UltraSPARC III Cu

La serie UltraSPARC è un'implementazione dell'architettura SPARC Version 9 realizzata da Sun. Dal punto di vista degli utenti e dei programmatori (cioè al livello ISP) i vari modelli sono molto simili e le differenze riguardano principalmente le prestazioni e il prezzo, ma a livello di microarchitettura differiscono in maniera considerevole. In questo paragrafo descriveremo il processore UltraSPARC III Cu. Il termine Cu[1] presente nel nome del processore indica l'uso di collegamenti in rame nel chip, al posto dell'alluminio utilizzato nei modelli precedenti. Il rame ha una conducibilità elettrica migliore dell'alluminio e permette la realizzazione di collegamenti più fini e una maggior velocità di funzionamento.

L'UltraSPARC III Cu è una macchina interamente a 64 bit, con registri e percorso dati a 64 bit; per ragioni di retrocompatibilità con l'architettura SPARC Version 8 (a 32 bit) è anche in grado di gestire operandi a 32 bit e può a tutti gli effetti eseguire software programmato per SPARC a 32 bit senza alcuna modifica. Anche se l'architettura interna è a 64 bit, il bus ha una larghezza di 128 bit, e in ciò è analogo al Pentium II, la cui architettura è a 32 bit, ma il cui bus è a 64 bit. In entrambi i casi il bus è una generazione più avanti rispetto alla CPU.

Diversamente dal Pentium 4, l'UltraSPARC è una vera e propria architettura RISC; ciò significa che non è necessario adottare per l'esecuzione un complicato meccanismo di conversione di vecchie istruzioni CISC in sequenze di micro-operazioni. Tuttavia nel corso degli anni sono state aggiunte istruzioni per la grafica e la multimedialità la cui esecuzione richiede la presenza di hardware specializzato.

Descrizione della microarchitettura di UltraSPARC III Cu

La Figura 4.48 mostra il diagramma a blocchi dell'UltraSPARC III Cu. Nel complesso è molto meno complicato della microarchitettura NetBurst del Pentium 4, perché deve implementare un'architettura ISA decisamente più semplice. Ciononostante alcuni dei componenti chiave sono simili a quelli del Pentium 4. Le somiglianze sono determinate principalmente dalla tecnologia disponibile e da fattori economici. Per esempio, all'epoca del progetto di questi chip le cache di dati di dimensione compresa tra 8 e 16 KB erano facilmente realizzabili oltre che convenienti e per questo motivo le ritroviamo in entrambe le architetture. Se un giorno le cache L1 da 64 MB dovessero diventare tecnologicamente facili da realizzare e, allo stesso tempo, economicamente convenienti, tutte le CPU le adotterebbero. Le differenze risiedono invece nella necessità, o meno, di dover realizzare un ponte tra un vecchio sistema d'istruzioni CISC e un moderno nucleo di tipo RISC.

Nella parte alta della Figura 4.48 si trova la cache delle istruzioni a 32 KB, di tipo associativo a 4 vie, che utilizza linee di cache da 32 byte. Dato che la maggior parte delle istruzioni UltraSPARC sono di 4 byte, questa cache può mantenere circa 8K istruzioni, leggermente meno della cache delle tracce di NetBurst.

L'unità di **rifornimento delle istruzioni** prepara fino a quattro istruzioni per ciclo per l'esecuzione. Nel caso in cui si verifichi un fallimento della cache L1, viene emesso un numero minore d'istruzioni. Quando si incontra un salto condizionale si consulta una **tabella dei salti** composta da 16 KB elementi per predire se occorre prelevare l'istruzione successiva o

[1] Cu è il simbolo chimico del rame (dal latino *cuprum*) (*N.d.T.*).

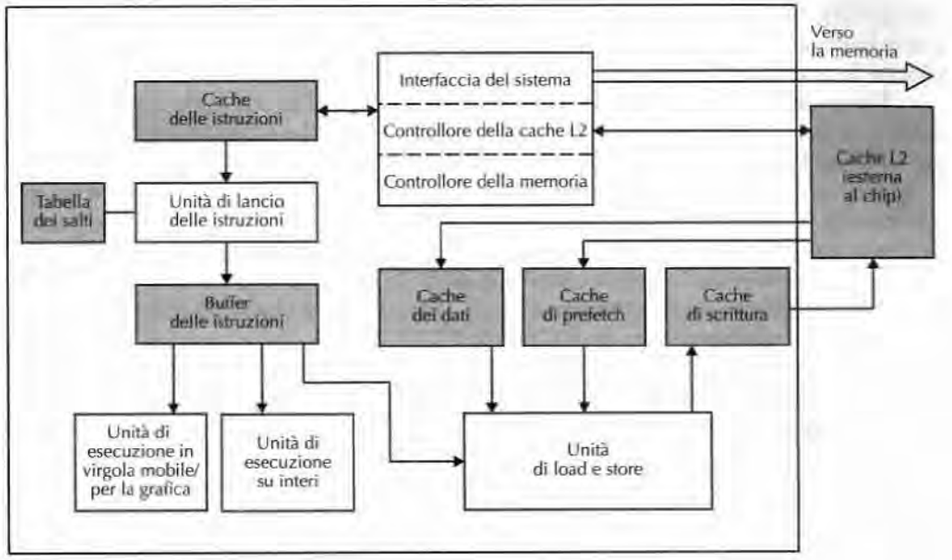


Figura 4.48 Diagramma a blocchi dell'UltraSPARC III Cu.

quella che si trova all'indirizzo di destinazione del salto. Un bit aggiuntivo associato a ciascuna parola aiuta inoltre a rendere più precisa la predizione dei salti. Le istruzioni, dopo essere state preparate, vengono inserite in un buffer a 16 posizioni il cui compito è quello di regolarizzare il flusso d'istruzioni entrante nelle pipeline.

Come mostra la Figura 4.48, l'output del buffer d'istruzioni passa attraverso l'unità di esecuzione degli interi, l'unità di esecuzione in virgola mobile e l'unità di caricamento e memorizzazione. L'unità di esecuzione degli interi contiene due ALU, oltre a una piccola pipeline per le istruzioni delle diramazioni. Al suo interno sono anche contenuti i registri ISA e alcuni registri di lavoro.

L'unità in virgola mobile contiene 32 registri e tre ALU distinte che servono, rispettivamente, per l'addizione/sottrazione, la moltiplicazione e la divisione. Al loro interno vengono eseguite anche le istruzioni grafiche.

L'unità di caricamento e memorizzazione gestisce varie istruzioni che si riferiscono alla memoria, e ha percorsi dati che utilizzano tre diverse cache. La **cache dei dati** è una tradizionale cache L1 da 64 KB di tipo associativo a 4 vie e con linee di 32 byte. La **cache di prefetch** da 2 KB è presente in quanto l'architettura ISA di UltraSPARC contiene istruzioni di prefetch che permettono al compilatore di prelevare parole di dati prima ancora che siano necessarie. Quando un compilatore ritiene che una certa parola potrebbe essere necessaria può emettere un'istruzione di prefetch, il cui effetto è quello di caricare in anticipo, all'interno della cache di prefetch, la linea indirizzata, in modo da velocizzarne l'accesso nel momento in cui, poche istruzioni dopo, verrà richiesta. In certe circostanze viene anche effettuato il prefetching hardware, al fine di migliorare le prestazioni dei programmi più vecchi che non prevedevano il prefetching. La **cache di scrittura** è pic-

cola (2 KB), ed è usata per raggruppare i dati da scrivere in memoria per sfruttare al meglio la larghezza (256 bit) del bus verso la cache L2. La sua unica funzione è quella di migliorare le prestazioni.

Il chip contiene anche i componenti logici per controllare l'accesso alla memoria, che sono divisi in tre parti: l'interaccia del sistema, il controllore della cache L2 e il controllore della memoria. L'interfaccia del sistema collega il chip alla memoria mediante un bus di 128 bit. Tutte le richieste verso il mondo esterno, tranne quelle verso la cache L2, passano attraverso quest'interfaccia. Dato che gli indirizzi fisici di memoria sono a 43 bit, in teoria sarebbe possibile avere una memoria centrale di 8×10^{12} byte; la dimensione del circuito stampato sul quale è montato il processore limita però la quantità massima di memoria a 16 GB. L'interfaccia è stata progettata per permettere la connessione di più UltraSPARC a una stessa memoria in modo da formare un **multiprocessore** (vedi Capitolo 8).

Il controllore della cache L2 permette di interfacciare il chip con la cache L2 unificata, posta all'esterno della CPU. Dato che la cache L2 è esterna al chip, essa può avere dimensioni di 1, 4 e 8 MB. La dimensione della linea dipende dalla dimensione della cache e varia da 64 byte, per una cache a 1 MB, fino a 512 byte, per una cache a 8 MB. Al contrario, la cache L2 del Pentium 4 si trova sul chip ed è limitata a un massimo di 1 MB per problemi di spazio. La scelta architetturale di UltraSPARC ha portato a una cache con una frequenza di successi sensibilmente maggiore rispetto a quella del Pentium 4 (dato che la cache è più grande), ma con un tempo di accesso più lento (dato che la cache è esterna al chip).

Il controllore della memoria mappa indirizzi virtuali a 64 bit in indirizzi fisici a 43 bit. L'UltraSPARC supporta la memoria virtuale (trattata nel Capitolo 6) con pagine di 8, 64, 512 KB e 4 MB. Per accelerare la corrispondenza ci sono delle tabelle speciali, chiamate **TLB** (*Translation Lookaside Buffers*), per confrontare l'indirizzo virtuale al quale si sta facendo riferimento con quelli referenziati nel passato più recente. Sono presenti tre di queste tabelle per gestire in modo flessibile le differenti dimensioni delle pagine e altre due per effettuare la corrispondenza delle istruzioni.

Pipeline dell'UltraSPARC III Cu

L'UltraSPARC III Cu dispone di una pipeline a 14 stadi. La Figura 4.49 ne mostra una rappresentazione semplificata, in cui i 14 stadi sono indicati, sul lato sinistro, dalle lettere comprese tra *A* e *D*. Esaminiamola brevemente. Lo stadio *A* (*Address generation*) si trova nella parte iniziale della pipeline ed è il punto in cui si determina l'indirizzo della successiva istruzione da prelevare. Di solito questo indirizzo corrisponde a quello che segue l'istruzione corrente. Tuttavia questo ordinamento sequenziale può essere alterato per varie ragioni; per esempio quando l'istruzione precedente è una diramazione che secondo la predizione deve essere eseguita, oppure quando occorre gestire un interrupt o un'eccezione. Dato che non è possibile effettuare la predizione dei salti in un solo ciclo, l'istruzione che segue un salto condizionale viene sempre eseguita, indipendentemente dal fatto che la diramazione verrà o meno effettuata.

Lo stadio *P* (*Preliminary fetch*) usa l'indirizzo fornito dallo stadio *A* per iniziare a prelevare fino a quattro istruzioni per ciclo dalla cache L1 delle istruzioni. In questa fase viene anche consultata la tabella dei salti per controllare se qualcuna di queste istruzioni è una diramazione condizionale e, nel caso lo sia, se la predizione indica di effettuare il salto. Lo stadio *F* (*Fetch*) completa il prelievo delle istruzioni dalla cache.

Lo stadio *B* (*Branch target*) decodifica le istruzioni appena prelevate. Nel caso in cui qualcuna di loro sia un salto condizionale che va effettuato, l'istruzione viene passata alle stadio *A* affinché in futuro sia possibile prelevare l'istruzione direttamente.

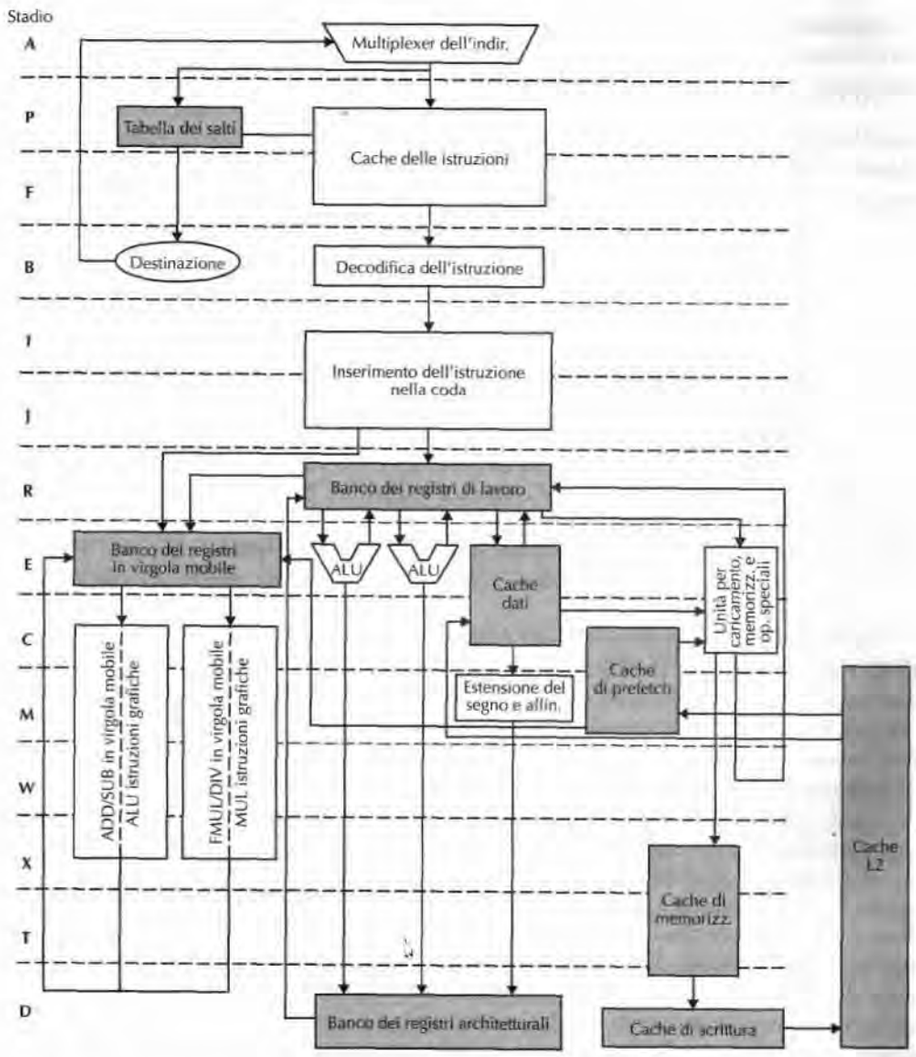


Figura 4.49 Rappresentazione semplificata della pipeline dell'UltraSPARC III Cu.

Lo stadio *I* (*Instruction group formation*) raggruppa le istruzioni in arrivo in base alle unità funzionali che devono utilizzare:

1. ALU degli interi, 1
2. ALU degli interi, 2
3. ALU in virgola mobile e per le istruzioni grafiche, 1
4. ALU in virgola mobile e per le istruzioni grafiche, 2
5. pipeline delle diramazioni (non mostrata nella figura)
6. operazioni di caricamento, di memorizzazione e altre di tipo speciale.

Le due ALU intere non sono perfettamente identiche e anche quelle in virgola mobile presentano tra loro notevoli differenze. In entrambi i casi ciò che distingue un'unità dall'altra sono le istruzioni che possono eseguire. Nello stadio *I* le istruzioni vengono ordinate in base alle unità che devono utilizzare.

Lo stadio *J* (*Instruction grouping*) rimuove le istruzioni dalla coda e si prepara a inviarle alle unità di esecuzione, cosa che avviene nel ciclo successivo. Durante ogni ciclo è possibile spostare nello stadio *R* fino a quattro istruzioni. La scelta delle istruzioni è vincolata dalla disponibilità delle unità funzionali. È possibile per esempio emettere contemporaneamente due istruzioni su interi, due istruzioni in virgola mobile, un'istruzione di caricamento e una di memorizzazione, mentre non è possibile emettere in un unico ciclo tre istruzioni su interi.

Lo stadio *R* cerca i registri richiesti dalle istruzioni su interi, mentre per le istruzioni in virgola mobile delega la richiesta all'insieme dei registri in virgola mobile. In questa fase vengono anche effettuati i controlli sulle dipendenze. Se un'istruzione richiede un registro che è ancora utilizzato da un'istruzione precedente, l'istruzione che ha effettuato la richiesta viene messa in stallo e vengono bloccate anche tutte quelle che la seguono. Diversamente dal Pentium 4, l'UltraSPARC III Cu non emette mai istruzioni fuori sequenza.

Nello stadio *E* (*Execution*) le istruzioni su interi vengono effettivamente eseguite. La maggior parte delle istruzioni aritmetiche, booleane e di scorrimento usa le ALU intere e la loro esecuzione è completata in un singolo ciclo. Al termine dell'esecuzione ogni istruzione aggiorna immediatamente il banco dei registri di lavoro. Alcune delle istruzioni più complesse vengono invece dirottate verso un'unità speciale. In questa fase iniziano anche, ma non terminano, le istruzioni di caricamento e memorizzazione. Inoltre vengono prelevati dal banco dei registri in virgola mobile gli operandi in virgola mobile. Sempre durante questo stadio si elaborano le istruzioni di salto condizionale e si determina se occorre oppure no effettuare la diramazione. Se la predizione si rivela errata viene spedito un segnale allo stadio *A* e la pipeline viene svuotata.

Durante lo stadio *C* (*Cache*) vengono completati gli accessi alla cache L1. In questa fase le istruzioni che leggono la memoria ottengono i propri dati.

Lo stadio *M* (*Miss*) inizia l'elaborazione delle parole di dati che sono state richieste, ma che non erano presenti nella cache L1. Viene effettuata una ricerca nella cache L2 e, se anche questa fallisce, viene emesso un riferimento alla memoria, che richiede un numero di cicli elevato. In questa fase vengono elaborati tutti i byte, i quarti di parola e le semi-parole che sono stati prelevati dalla cache L1 e che devono essere allineati oppure estesi con segno. Inoltre ottengono i propri dati anche le load in virgola mobile che possono essere soddisfatte dalla cache. Per i dati interi invece non si utilizza la cache di prefetch per via di complicate ragioni legate alla temporizzazione.

Nello stadio *W* (*Write*) i risultati prodotti dall'unità speciale vengono scritti nel banco dei registri di lavoro.

La maggior parte delle istruzioni in virgola mobile e di quelle grafiche termina nello stadio *X* (*eXtend*). I risultati vengono resi disponibili alle istruzioni successive attraverso l'avanzamento store-to-load prima ancora che le istruzioni siano formalmente ritirate durante lo stadio *D*.

Lo stadio *T* (*Trap*) rileva tutte le eccezioni e rende precise le eccezioni e gli interrupt; in altre parole deve salvare lo stato della macchina in cui tutte le istruzioni precedenti un'eccezione o un interrupt sono state interamente completate e nessuna di quelle successive ha avuto inizio.

Lo stadio *D* copia i registri degli interi e dei valori in virgola mobile nei loro rispettivi banchi di registri architetturali. Se si verifica un'eccezione o un interrupt vengono resi visibili questi valori e non quelli presenti nei registri di lavoro. La memorizzazione nei registri architetturali ha la stessa funzione dell'azione di ritiro nel Pentium. Inoltre, durante lo stadio *D*, tutte le istruzioni di memorizzazione completano la scrittura dei loro risultati nella cache di scrittura, invece che nella cache L1. Infine le linee di questa cache vengono scritte nella cache L2 senza passare attraverso la cache L1 (il cui contenuto è disgiunto da quello della cache L2). Questa particolare organizzazione delle cache mira a facilitare la realizzazione di multiprocessori UltraSPARC.

Questa descrizione dell'UltraSPARC è lontana dall'essere completa, ma permette di avere un'idea sostanzialmente corretta del suo funzionamento e delle differenze che la contraddistinguono rispetto alla microarchitettura del Pentium 4.

4.6.3 Microarchitettura della CPU 8051

Il nostro ultimo esempio è la microarchitettura dell'8051 (Figura 4.50) che è considerevolmente più semplice delle due precedenti. Il motivo risiede nelle piccole dimensioni del chip (60.000 transistor) e dal fatto che è stata progettata molto tempo prima che si diffondesse l'utilizzo delle pipeline. Inoltre uno dei principali obiettivi della progettazione di questo chip era di renderlo economico, non veloce. Economico e Semplice sono sempre dei buoni amici. Economico e Veloce invece non sempre lo sono.

Il cuore dell'8051 è rappresentato dal bus principale cui sono collegati vari registri, quasi tutti destinati a lettura o scrittura, sotto il controllo del programma. Descriviamoli brevemente. Il registro ACC è l'**accumulatore**, cioè il principale registro nel quale vengono memorizzati i risultati delle computazioni; la maggior parte delle istruzioni aritmetiche fa uso di questo registro. Il registro B è utilizzato per la moltiplicazione e la divisione, oltre che come deposito per risultati temporanei. Il registro SP è il puntatore allo stack e, come nella maggior parte delle macchine, punta alla sua cima. Il registro IR è il **registro istruzioni**, che contiene l'istruzione in corso di esecuzione.

I registri TMP1 e TMP2 sono due latch collegati alla ALU. Per eseguire un'operazione, prima di far partire la ALU, gli operandi vengono copiati in questi latch. L'output della ALU può essere memorizzato in uno qualsiasi dei registri scrivibili utilizzando il bus principale. Inoltre i codici di condizione, che indicano se il risultato era zero, negativo, e così via, vengono scritti nel registro PSW (*Program Status Word*).

L'8051 è dotato di due memorie separate per codice e dati. La RAM dei dati è di soli 128 byte (ma l'8052 ne ha 256) e quindi gli 8 bit del registro RAM ADDR sono sufficienti per indirizzarla. Per accedere alla RAM si copia l'indirizzo del byte desiderato nel registro RAM ADDR e poi si dà inizio all'operazione di memoria. La memoria del codice può raggiungere i 64 KB (utilizzando memoria esterna al chip) e quindi il registro ROM ADDR utilizzato per indirizzarla deve avere almeno 16 bit. Questo registro ROM ADDR indirizza il codice del programma nella ROM, in modo analogo a quanto abbiamo visto per la RAM.

DPTR (*Double PoinTeR*) è un registro di lavoro a 16 bit per gestire e assemblare gli indirizzi a 16 bit. Il registro PC è il solito **contatore d'istruzioni** e il registro PC INCREMENTER è un circuito speciale che si comporta come uno pseudo-registro. Quando si copia PC al suo interno e poi lo si legge, il valore risulta automaticamente incrementato di uno. Né il PC né PC INCREMENTER sono raggiungibili dal bus principale. Infine BUFFER è un altro registro di lavoro a 16 bit. Tutti questi registri consistono in realtà in coppie di registri a 8 bit che possono essere manipolati indipendentemente.

Oltre al bus principale e ai registri appena descritti, sul chip dell'8051 ci sono tre timer a 16 bit che risultano indispensabili per le applicazioni in tempo reale. Sono presenti anche quattro porte di I/O a 8 bit, che permettono all'8051 di controllare 32 dispositivi esterni come pulsanti, luci, sensori, attuatori. È soprattutto la presenza dei timer e delle porte di I/O che, più di ogni altro aspetto, rende possibile l'utilizzo dell'8051 nelle applicazioni integrate senza aver bisogno di chip aggiuntivi.

L'8051 è un processore sincrono in cui la maggior parte delle istruzioni, seppur non tutte, impiegano un solo ciclo di clock. Ogni ciclo di clock può essere suddiviso in sei parti, chiamate **stati**. Durante il primo stato la nuova istruzione viene prelevata dalla ROM, inserita nel bus principale e inviata al registro IR. Nel secondo stato l'istruzione viene decodificata e PC viene incrementato. Nel successivo vengono preparati gli operandi e nel quarto uno degli operandi viene portato sul bus principale (di solito viene memorizzato nel registro TMP da dove può essere utilizzato come operando della ALU). Sempre durante questo stato il registro ACC può essere copiato in TMP2; in questo modo entrambi gli operandi della ALU sono pronti per essere utilizzati. Durante il quinto stato la ALU compie la propria esecuzione. Infine, durante il sesto stato, l'output della ALU viene scritto all'interno della sua destinazione utilizzando il bus principale. Nello stesso momento il registro ROM ADDR viene impostato in modo da prelevare l'istruzione successiva.

Anche se potremmo fornire maggiori dettagli su questo processore, la descrizione data in questo paragrafo e la Figura 4.50 sono sufficienti per avere un'idea di base della sua microarchitettura. L'8051 ha un unico bus principale (per ridurre l'area del chip) al quale sono collegati un insieme eterogeneo di registri, tre timer e quattro porte di I/O, oltre ad alcuni registri aggiuntivi su un bus locale. Durante ogni ciclo del percorso due operandi vengono fatti passare attraverso la ALU e il risultato viene memorizzato all'interno di un registro, proprio come nei calcolatori moderni.

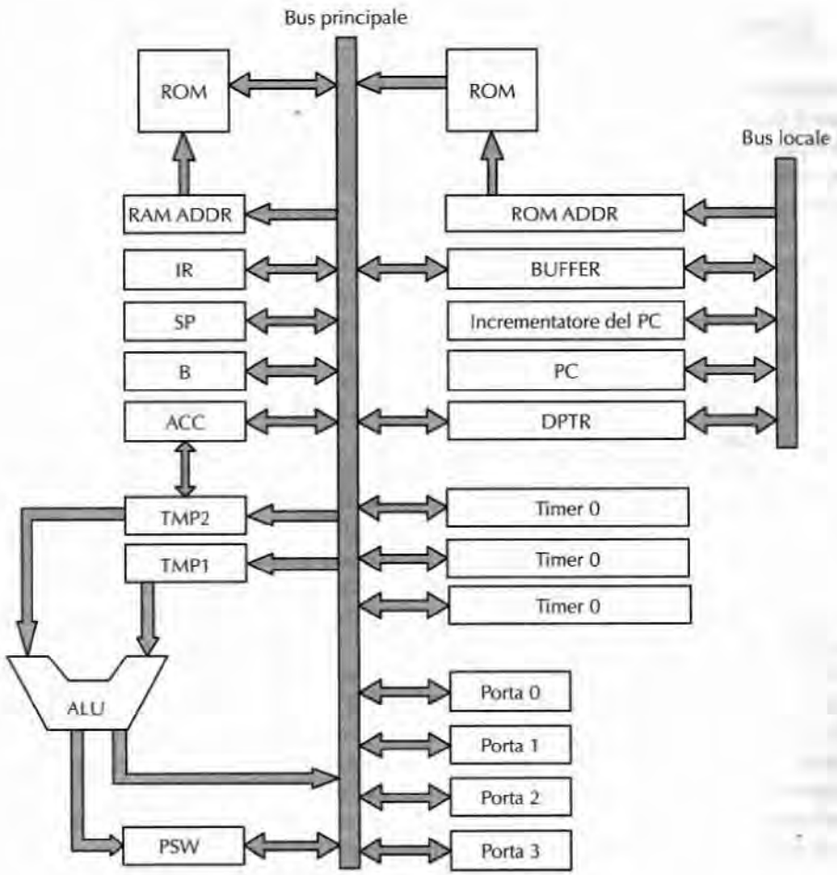


Figura 4.50 Microarchitettura dell'8051.

4.7 Confronto tra Pentium, UltraSPARC e 8051

I tre esempi considerati sono molto differenti tra loro, pur avendo dei punti in comune. Il Pentium 4 è un insieme d'istruzioni CISC datato che gli ingegneri di Intel amerebbero fervidamente poter gettare nella Baia di San Francisco, se ciò non violasse la legge della California sull'inquinamento delle acque. L'UltraSPARC III è una vera e propria architettura RISC, con un insieme d'istruzioni semplice e ordinato. L'8051 è invece un semplice processore a 8 bit per le applicazioni integrate. Nel cuore di ognuno ci sono dei registri e una o più ALU che eseguono semplici operazioni aritmetiche e booleane.

Nonostante le ovvie differenze esterne il Pentium 4 e l'UltraSPARC III hanno delle unità di esecuzione abbastanza simili. Le unità di esecuzione di entrambi accettano micro-operazioni che specificano un codice operativo, due registri sorgenti e un registro destinazione. Esse sono in grado di eseguire in ciascun ciclo una micro-operazione; hanno inoltre pipeline con un elevato numero di stadi, effettuano la predizione dei salti e sono dotate di cache separate per i dati e per le istruzioni.

Queste analogie tra i componenti interni non sono casuali e neanche dovuti all'interminabile passaggio degli ingegneri della Silicon Valley da una società a un'altra. Come abbiamo visto nelle architetture di Mic-3 e Mic-4, è facile e naturale costruire un percorso dati a pipeline che prende due registri sorgente, li fa passare attraverso la ALU e ne memorizza il risultato nuovamente in un registro. Nella Figura 4.34 è possibile vedere una rappresentazione grafica di una pipeline di questo tipo. Con la tecnologia attualmente disponibile, questa architettura è la più efficiente.

La differenza principale tra il Pentium 4 e l'UltraSPARC III è nel modo in cui prendono le istruzioni del loro livello ISA e le assegnano alle unità funzionali. Il Pentium 4 deve suddividere le sue istruzioni CISC per portarle nel formato a tre registri richiesto dall'unità funzionale. Questo è ciò che fa il front end della Figura 4.47: trasforma istruzioni grandi in micro-operazioni dal formato semplice e regolare. L'UltraSPARC III non deve effettuare questa conversione, dato che le sue istruzioni native hanno già un formato semplice e regolare. Questo è il motivo per cui la maggior parte dei nuovi ISA sono di tipo RISC: permettono una miglior corrispondenza tra le istruzioni ISA e l'unità di esecuzione interna.

Può essere istruttivo confrontare la nostra architettura finale, il Mic-4, con i due esempi tratti dal mondo reale. Il Mic-4 assomiglia al Pentium 4, dato che entrambi devono interpretare un insieme d'istruzioni ISA di tipo non-RISC. Per far ciò entrambi suddividono le istruzioni ISA in micro-operazioni composte da un codice operativo, due registri sorgenti e un registro destinazione. In entrambi i casi le micro-operazioni vengono accodate prima di essere eseguite. Mic-4 emette le istruzioni, le esegue e le ritira sempre in ordine, mentre il Pentium 4 utilizza una politica di emissione in ordine, esecuzione fuori sequenza e ritiro in ordine.

Mic-4 e UltraSPARC non sono effettivamente confrontabili, dato che l'insieme d'istruzioni ISA dell'UltraSPARC è composto da istruzioni RISC (cioè micro-operazioni a tre registri) che non devono essere suddivise. Ognuna di queste istruzioni può essere eseguita, senza alcuna modifica, in un unico ciclo del percorso dati.

Tuttavia, diversamente dal Pentium 4 e dall'UltraSPARC III, l'8051 è una macchina molto semplice. È più di tipo RISC che di tipo CISC, dato che la maggior parte delle sue istruzioni può essere eseguita in un singolo ciclo di clock e non deve essere suddivisa. Non sono presenti né pipeline né cache; inoltre l'emissione, l'esecuzione e il ritiro avvengono tutte in ordine. Nella sua semplicità assomiglia a Mic-1.

4.8 Riepilogo

Il cuore di ogni calcolatore è rappresentato dal percorso dati, che contiene dei registri, uno, due o tre bus, e una o più unità funzionali come le ALU e gli shifter. Il ciclo di esecuzione principale consiste nel prelevare operandi dai registri e nello spedirli, utilizzando i bus, alle ALU e alle altre unità funzionali per l'esecuzione. I risultati vengono poi memorizzati nuovamente nei registri.

Il percorso dati può essere controllato da un sequenzializzatore che preleva le microistruzioni da una memoria di controllo. Ogni microistruzione contiene dei bit che controllano il percorso dati durante un ciclo e specificano quali operandi selezionare, quale operazione eseguire e che cosa occorre fare con il risultato. Inoltre, ogni microistruzione specifica il proprio successore, di solito in modo esplicito, indicandone l'indirizzo. Alcune microistruzioni modificano questo indirizzo di base calcolandone l'OR con alcuni bit prima di utilizzarlo.

La macchina IJVM è una macchina a stack con codici operativi da 1 byte che permettono di usare lo stack per inserire, estrarre e combinare (cioè addizionare) parole. Nella microarchitettura Mic-1 è stata presentata un'implementazione microprogrammata. Aggiungendo un'unità per il prelievo delle istruzioni, in grado di caricare in anticipo i byte del flusso delle istruzioni, è possibile eliminare un gran numero di riferimenti al contatore d'istruzioni, accelerando notevolmente la macchina.

Esistono diversi modi per progettare il livello di microarchitettura. Occorre compiere molte scelte, come quella tra un'architettura a due bus e una a tre bus, tra i campi codificati oppure decodificati delle microistruzioni, tra l'utilizzo o meno del prefetching, sulla profondità delle pipeline, e molte altre ancora. Mic-1 è una semplice macchina controllata via software, in cui l'esecuzione è sequenziale e non vi è parallelismo. Al contrario Mic-4 è una microarchitettura altamente parallela con una pipeline a sette stadi.

Le prestazioni possono essere migliorate agendo sotto diversi aspetti della microarchitettura. Il modo principale per migliorare le prestazioni consiste nell'uso di memorie cache. Per accelerare i riferimenti alla memoria si usano comunemente cache a corrispondenza diretta e cache set-associative. Anche la predizione dei salti, sia statica sia dinamica, è importante, così come l'esecuzione fuori sequenza e l'esecuzione speculativa.

Le tre macchine di esempio, Pentium 4, UltraSPARC III e 8051, hanno una microarchitettura invisibile ai programmatori del linguaggio assemblativo ISA. Il Pentium 4 utilizza uno schema complesso per convertire le istruzioni ISA in micro-operazioni, memorizzarle in una cache e alimentare con loro un nucleo RISC superscalare. Questo schema permette inoltre l'esecuzione fuori sequenza, la rinomina dei registri e tutti gli altri meccanismi descritti nel libro che consentono di ottenere anche un ultimo, minimo, miglioramento della velocità dell'hardware. L'UltraSPARC III Cu ha una pipeline profonda, ma a parte ciò è relativamente semplice, emettendo, eseguendo e ritirando le istruzioni in ordine. L'8051 è decisamente semplice: è composto da un singolo bus principale al quale sono collegate una manciata di registri e una ALU.

Problemi

1. Nella Figura 4.6 il registro del bus B è codificato in un campo a 4 bit, mentre il bus C è rappresentato come una mappa di bit. Perché?
2. Nella Figura 4.6 è rappresentato un rettangolo etichettato come "Bit alto". Si disegni il circuito che lo realizza.
3. Quando viene abilitato il campo JMPC di una microistruzione, viene calcolato l'OR logico tra MBR e NEXT_ADDRESS, per formare l'indirizzo della microistruzione successiva. Esistono situazioni in cui ha senso che il campo NEXT_ADDRESS valga 0x1FF e che si utilizzi JMPC?
4. Supponiamo che nell'esempio della Figura 4.14(a) si aggiunga l'istruzione

   ```
   k = 5;
   ```

   dopo l'istruzione `if`. Quale dovrebbe essere il nuovo codice assemblativo? Si assuma che il compilatore sia ottimizzato.
5. Si forniscano due differenti traduzioni IJVM della seguente istruzione Java:

   ```
   i = k + n + 5;
   ```

6. Si scriva l'istruzione Java che produce il seguente codice IJVM:

   ```
   ILOAD j
   ILOAD n
   ISUB
   BIPUSH 7
   ISUB
   DUP
   IADD
   ISTORE i
   ```

7. Nel testo abbiamo affermato che quando si traduce in binario l'istruzione

   ```
   if (z) goto L1; else goto L2
   ```

   L2 deve trovarsi nelle prime 256 parole della memoria di controllo. Non sarebbe stato ugualmente possibile avere, per esempio, *L1* all'indirizzo 0x40 e *L2* all'indirizzo 0x140? Si motivi la risposta.
8. Nel microprogramma Mic-1, MDR viene copiato all'interno di H, durante `if_icmpeq3`. Dopo qualche linea viene sottratto da TOS per effettuare un test di uguaglianza. Sicuramente sarebbe stato meglio avere in questo punto la seguente istruzione:

   ```
   If_cmpeq3 Z = TOS - MDR; rd
   ```

   Perché ciò non è stato fatto?
9. Si calcoli quanti nanosecondi impiega Mic-1 a 2,5 GHz per eseguire la seguente istruzione Java

   ```
   i = j + k;
   ```

10. Si risponda nuovamente alla domanda precedente considerando questa volta Mic-2 a 2,5 GHz. In base al calcolo effettuato quanto dovrebbe impiegare su Mic-2 un programma che viene eseguito da Mic-1 in 100 s?
11. Si scriva il microcodice che implementa per Mic-1 l'istruzione IJVM POPTWO (che rimuove due parole dalla cima dello stack).

12. Su una macchina JVM completa esistono codici operativi speciali per caricare nello stack, senza utilizzare l'istruzione generale ILOAD, le variabili locali da 0 a 3. Come dovrebbe essere modificata IJVM per sfruttare nel modo migliore queste istruzioni?
13. L'istruzione ISHR (scorrimento aritmetico a destra su interi) esiste in JVM, ma non in IJVM. Essa utilizza i due valori in cima allo stack, sostituendoli con un unico valore, il risultato. La seconda parola a partire dalla cima dello stack è l'operando da traslare. Il suo contenuto è spostato a destra di un valore compreso tra 0 e 31; questo valore è specificato nei 5 bit meno significativi della parola in cima allo stack (gli altri 27 bit sono invece ignorati). Il bit del segno viene replicato a destra per tanti bit quanti sono quelli da spostare. Il codice operativo di ISHR è 122 (0x7A).
    a. Qual è l'operazione aritmetica che equivale a traslare a destra di un valore 2?
    b. Si estenda il microcodice per includere questa istruzione in IJVM.
14. L'istruzione ISHL (scorrimento a sinistra su interi) esiste in JVM, ma non in IJVM. Essa utilizza i due valori in cima allo stack, sostituendoli con un unico valore, il risultato. La seconda parola a partire dalla cima dello stack è l'operando da traslare. Il suo contenuto è spostato a destra di un valore compreso tra 0 e 31; questo valore è specificato nei 5 bit meno significativi della parola in cima allo stack (gli altri 27 bit sono invece ignorati). Sulla destra vengono inseriti tanti valori zero quanti sono i bit da spostare. Il codice operativo di ISHL è 120 (0x78).
    a. Qual è l'operazione aritmetica che equivale a traslare a sinistra di un valore 2?
    b. Si estenda il microcodice per includere questa istruzione in IJVM.
15. L'istruzione JVM INVOKEVIRTUAL deve conoscere il numero dei suoi parametri. Perché?
16. Si implementi per Mic-2 l'istruzione JVM DLOAD. Essa ha un indice a 1 byte e inserisce nello stack la variabile locale che si trova in questa posizione. Successivamente inserisce nello stack anche la parola che si trova nell'indirizzo successivo, più in alto.
17. Si disegni un automa a stati finiti per gestire il punteggio di una partita di tennis. Le regole del tennis sono le seguenti. Per vincere occorrono almeno quattro punti e occorre avere almeno due punti più dell'avversario. Si inizia nello stato (0, 0), che indica che non è stato ancora attribuito alcun punteggio. Si aggiunga quindi uno stato (1, 0) che indica che A ha segnato un punto; si etichetti con *A* l'arco che collega (0, 0) con (1, 0). Si aggiunga poi uno stato (0, 2) che indica che *B* ha segnato un punto e si etichetti con *B* l'arco proveniente da (0, 0). Si continui ad aggiungere stati e archi finché non siano stati inclusi tutti i possibili stati.
18. Si consideri nuovamente il problema precedente. Ci sono degli stati che potrebbero essere condensati in un unico stato senza cambiare il risultato delle partite? Se sì, quali sono?
19. Si disegni un automa a stati finiti per la predizione dei salti che sia più tenace di quello della Figura 4.42 nel senso di modificare le predizioni soltanto dopo tre previsioni errate consecutive.
20. Il registro di scorrimento della Figura 4.27 ha una capacità massima di 6 byte. Si potrebbe costruire una versione più economica della IFU con un registro di scorrimento a 5 byte? E che cosa si può dire di una versione con registro a 4 byte?
21. Dopo aver esaminato, nell'esercizio precedente, versioni più economiche della IFU, consideriamo ora quelle più costose. Ci potrebbe essere una qualche ragione per avere registri di scorrimento più grandi, per esempio di 12 byte? Si motivi la risposta.
22. Nel microprogramma Mic-2, il codice dell'istruzione `if_icmpeq6` porta a T quando Z è impostato a 1. Il codice in T è tuttavia identico a quello in `goto1`. Sarebbe stato possibile andare direttamente in `goto1`? Così facendo la macchina sarebbe risultata più veloce?

23. In Mic-4 l'unità di decodifica fa corrispondere al codice operativo IJVM un indice della ROM dove sono memorizzate le micro-operazioni corrispondenti. Potrebbe sembrare più semplice omettere semplicemente la fase di decodifica e inserire direttamente il codice operativo IJVM nella coda. Si potrebbe usare il codice operativo di IJVM come un indice all'interno della ROM, allo stesso modo di Mic-1. Che cosa c'è di errato in questa strategia?

24. Un calcolatore ha due livelli di cache. Supponiamo che il 60% dei riferimenti alla memoria sia soddisfatto dalla cache di primo livello, il 35% da quella di secondo livello e il 5% fallisca con entrambe le cache. I tempi di accesso sono rispettivamente di 5 ns, 15 ns e 60 ns. I tempi relativi alla cache di secondo livello e alla memoria iniziano a partire dal momento in cui si sa che è necessario accedere a queste memorie (per esempio, un accesso alla cache di secondo livello non comincia prima di un fallimento della cache di primo livello). Qual è il tempo medio di accesso?

25. Alla fine del Paragrafo 4.5.1 abbiamo detto che la tecnica della *write allocation* è vantaggiosa solo se è probabile che in una riga ci siano più scritture nella stessa linea. Che cosa si può dire riguardo il caso in cui una scrittura sia seguita da più letture?

26. Nella prima stesura di questo libro la Figura 4.39 mostrava una cache associativa a tre vie invece di una a quattro vie. Uno dei revisori del testo ha avuto un improvviso scatto d'ira, affermando che quella figura avrebbe confuso orribilmente gli studenti, dato che tre non è una potenza di due e i calcolatori eseguono ogni operazione in binario. Poiché il cliente ha sempre ragione la figura è stata modificata in una cache associativa a quattro vie. Questa persona aveva effettivamente ragione? Si argomenti la risposta.

27. Un calcolatore con una pipeline a cinque stadi tratta i salti condizionali ponendo la pipeline in stallo per tre cicli dopo aver rilevato una di queste diramazioni. Di quanto degrada le prestazioni questa strategia di mettere in stallo l'esecuzione, nel caso in cui il 20% di tutte le istruzioni siano salti condizionali? Si ignorino tutti gli altri motivi che possono mettere in stallo l'esecuzione, eccezion fatta per i salti condizionali.

28. Supponiamo che un calcolatore sia in grado di effettuare il prefetch di un massimo di 20 istruzioni. Tuttavia, in media, quattro di queste istruzioni sono salti condizionali, ciascuno dei quali può essere predetto correttamente nel 90% dei casi. Qual è la probabilità che il prefetching sia sulla strada giusta?

29. Supponiamo di voler cambiare la progettazione della macchina usata nella Figura 4.43, in modo da avere 16 registri al posto di 8. Modifichiamo quindi l'istruzione 6 in modo che usi R8 come propria destinazione. Che cosa succede nei cicli a partire dal ciclo 6?

30. Generalmente le dipendenze provocano dei problemi alle CPU con pipeline. Ci sono delle ottimizzazioni che possono essere fatte per gestire le dipendenze di tipo WAR e che possono migliorare realmente la situazione? Quali?

31. Si riscriva l'interprete di Mic-1, facendo però in modo che LV punti alla prima variabile locale invece che al puntatore di collegamento.

32. Si scriva un simulatore per una cache a corrispondenza diretta a 1 via. Il numero degli elementi e la dimensione della linea devono essere parametri della simulazione. Si facciano degli esperimenti con il simulatore e si commentino i dati ricavati.

Capitolo 5

Livello di architettura dell'insieme d'istruzioni

Questo capitolo tratta in dettaglio il livello di architettura dell'insieme d'istruzioni (ISA, *Instruction Set Architecture*). Tale livello si trova tra quello della microarchitettura e il sistema operativo, come raffigurato nella Figura 1.2. Storicamente, il livello ISA fu il primo a essere sviluppato, e costituiva infatti l'unico livello presente. Ancor oggi si usa riferirsi a questo livello come all'"architettura" di una macchina o altre volte (in modo improprio) come al "linguaggio assemblativo".

Il livello ISA ha una rilevanza particolare che lo rende importante per i progettisti di sistemi: costituisce l'interfaccia tra il software e l'hardware. Anche se, in linea di principio, sarebbe possibile disporre di un hardware che esegue direttamente programmi scritti in C, C++, Java o in altri linguaggi d'alto livello, non si tratta di una buona idea. Così facendo si perderebbe l'incremento di prestazioni garantito dalla compilazione rispetto all'interpretazione. Inoltre, per ragioni di praticità, è auspicabile che i computer siano capaci di eseguire codice scritto in più linguaggi, invece che in uno solo.

L'approccio prediletto dalla quasi totalità dei progettisti di sistemi è di partire da vari linguaggi d'alto livello per poi tradurli in una forma intermedia comune, il livello ISA, e quindi costruire l'hardware in grado di eseguire direttamente i programmi di livello ISA. Tale livello definisce l'interfaccia tra i compilatori e l'hardware ed è il linguaggio che entrambi possono comprendere. Le relazioni intercorrenti tra compilatori, livello ISA e hardware sono mostrate nella Figura 5.1.

In linea di principio, la fase di progettazione di una nuova macchina richiede la consultazione sia dei progettisti del compilatore, sia dei progettisti dell'hardware, al fine di individuare le caratteristiche desiderate per il livello ISA.

Se gli autori del compilatore richiedono alcune caratteristiche che gli ingegneri non possono implementare con costi contenuti (per esempio un'istruzione *branch-and-do-payroll*, "salta e contabilizza"), allora queste vengono escluse dal progetto. Allo stesso modo se i responsabili dell'hardware ideano una qualche nuova caratteristica ingegnosa che pretendono di implementare (per esempio una memoria in cui è velocissimo accedere alle locazioni il cui indirizzo è un numero primo), ma gli addetti al software non riescono a capire come scrivere del codice che la possa usare, la proposta rimarrà sulla carta. Dopo molte trattative e simulazioni emergerà e verrà infine implementato un livello ISA ottimizzato alla perfezione per il linguaggio di programmazione richiesto.

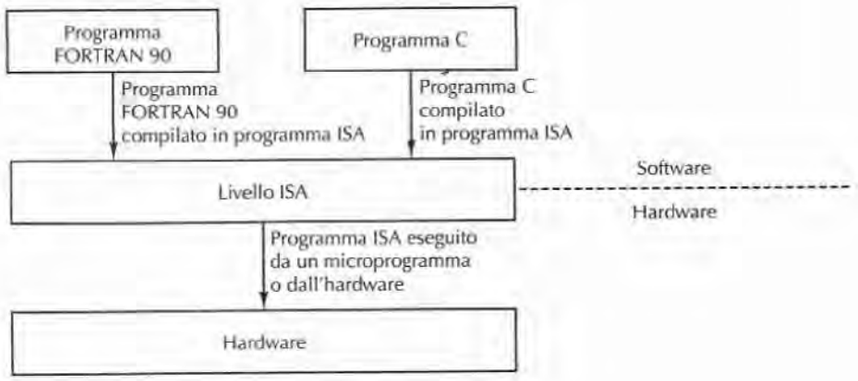


Figura 5.1 Il livello ISA è l'interfaccia tra compilatori e hardware.

Almeno in teoria. Nella bieca realtà, quando si tratta di sviluppare una nuova macchina la prima domanda posta dai potenziali utenti è: "il progetto è compatibile con il suo predecessore?". La seconda è: "posso farci girare il mio vecchio sistema operativo?". La terza è: "riuscirà a eseguire tutte le mie applicazioni senza bisogno di modificarle?". Qualora la risposta a una qualsiasi di queste domande fosse "no", i progettisti avrebbero dei seri problemi a giustificare il proprio lavoro. I clienti sono poco inclini a gettar via il loro vecchio software e ricominciare a scriverlo da capo.

Questo atteggiamento costringe gli architetti di computer a mantenere l'ISA costante di modello in modello, o a renderlo almeno **retrocompatibile**. Con ciò intendiamo che la nuova macchina debba essere in grado di eseguire i vecchi programmi senza modifiche. D'altra parte è del tutto comprensibile che una nuova macchina metta a disposizione alcune istruzioni e caratteristiche innovative che possono essere sfruttate solo dal software nuovo. Con riferimento alla Figura 5.1, i progettisti sono liberi di fare ciò che vogliono a livello hardware a patto che garantiscano un ISA retrocompatibile con i modelli precedenti; il livello hardware non interessa quasi a nessuno (e quasi nessuno sa come funziona). Così è possibile optare per un progetto di microprogrammazione o per l'esecuzione diretta, o aggiungere pipeline, funzionalità superscalari o qualunque altra cosa si desideri, ammesso che sia mantenuta la retrocompatibilità con l'ISA precedente. L'obiettivo è assicurare che i vecchi programmi girino sul nuovo processore e la sfida diventa la progettazione di macchine migliori soggette ai vincoli di retrocompatibilità.

Questo non vuol dire che la progettazione ISA conti poco; un buon ISA presenta vantaggi significativi rispetto a un cattivo progetto, specie in termini di potenza grezza di calcolo a parità di costi. Progetti ISA altrimenti equivalenti possono differire anche del 25% in termini di prestazioni. Intendiamo affermare semplicemente che il mercato rende difficile (se non impossibile) l'abbandono di un ISA datato per introdurne uno nuovo. Ciononostante, di tanto in tanto emerge un nuovo ISA per uso generale, il che capita molto più spesso nei settori di mercato specializzati (per esempio nella progettazione di sistemi integrati o di processori multimediali). Per queste ragioni è importante comprendere la progettazione ISA.

Che cosa rende un ISA un "buon ISA"? Ci sono due fattori principali. Innanzitutto un buon ISA dovrebbe definire un insieme d'istruzioni che può essere implementato efficientemente da tecnologie presenti e future, il che risulta in progetti duraturi e dunque economicamente vantaggiosi. Una cattiva progettazione è più difficile da implementare e potrebbe richiedere molte più porte logiche per realizzare il processore, nonché più memoria per eseguirne i programmi. Potrebbe inoltre rallentare l'esecuzione perché offuscherebbe la possibilità di sovrapporre l'esecuzione di alcune istruzioni, richiedendo espedienti molto sofisticati per raggiungere prestazioni equivalenti. Un progetto che trae vantaggio dalle peculiarità di una particolare tecnologia può rivelarsi un fuoco di paglia e fornire un sola generazione d'implementazioni economicamente vantaggiose, per poi essere sorpassato da ISA più lungimiranti.

In secondo luogo, un buon ISA dovrebbe favorire una compilazione del codice "pulita". La regolarità e la completezza del ventaglio di scelte disponibili per il compilatore costituiscono un tratto importante che non sempre gli ISA rispettano. Si tratta di proprietà importanti per il compilatore, che altrimenti potrebbe trovarsi in difficoltà nell'operare la scelta migliore tra alternative limitate, in special modo quando alcune scelte apparentemente ovvie non sono permesse dall'ISA. In breve, dato che l'ISA è l'interfaccia tra hardware e software, dovrebbe soddisfare sia i progettisti hardware (se facile da implementare efficientemente), sia i progettisti software (se favorevole alla produzione di codice di qualità).

5.1 Panoramica del livello ISA

Per cominciare lo studio del livello ISA chiediamoci che cos'è. Potrebbe sembrare una domanda semplice a cui rispondere, ma solleva più complicazioni di quanto si potrebbe immaginare a prima vista. Qui di seguito trattiamo alcune questioni controverse, dopodiché daremo uno sguardo a modelli di memoria, registri e istruzioni.

5.1.1 Proprietà del livello ISA

In linea di principio, il livello ISA si può definire come l'aspetto che la macchina assume agli occhi di un programmatore in linguaggio macchina. Siccome oramai nessun programmatore (sensato) programma più in linguaggio macchina, ridefiniamo il concetto dicendo che il codice di livello ISA è l'output di un compilatore (senza tener conto, per il momento, delle chiamate di sistema operativo e del linguaggio assemblativo simbolico). Al fine di produrre codice di livello ISA, il progettista del compilatore deve conoscere il modello di memoria, quali registri ci sono, quali sono i tipi di dati e d'istruzioni disponibili, e così via. L'insieme di tutte queste informazioni definisce il livello ISA.

Secondo questa definizione, il fatto che la microarchitettura sia o meno microprogrammata (o che disponga di pipeline, o che sia superscalare, e così via) non fa parte del livello ISA, perché non è visibile al progettista del compilatore. In realtà questa osservazione non è del tutto corretta, poiché alcune di queste proprietà influenzano le prestazioni, il che è visibile a chi scrive il compilatore. Si consideri, per esempio, un progetto superscalare che prevede l'emissione di due istruzioni in successione immediata all'interno dello stesso ciclo se un'istruzione è di tipo intero e l'altra in virgola mobile. Se il compilatore

alternasse istruzioni intere a istruzioni in virgola mobile otterrebbe prestazioni visibilmente migliori che non in caso contrario. Così i dettagli dell'operazione superscalare *sono* visibili a livello ISA, e quindi la separazione trà i livelli non è così chiara come potrebbe apparire inizialmente.

Per alcune architetture, ma non sempre, il livello ISA è specificato attraverso un documento formale di definizione, spesso redatto da un consorzio di aziende. Per esempio la versione 9 dello SPARC ha una definizione ufficiale (Weaver e Germond, 1994). Lo scopo di un documento di definizione è di mettere i diversi produttori in grado di costruire macchine capaci di eseguire lo stesso codice, la cui esecuzione dia esattamente gli stessi risultati.

Nel caso dello SPARC, l'idea è di consentire ai diversi fornitori di chip di produrre chip SPARC che funzionino tutti in modo identico, differendo solo per costi e prestazioni. Perché l'idea funzioni, è necessario che i produttori di chip conoscano il funzionamento di un chip SPARC (a livello ISA). Di conseguenza il documento di definizione specifica il modello di memoria, i registri presenti, il comportamento delle istruzioni e così via, ma non la microarchitettura.

Tali documenti di definizione contengono **sezioni normative**, che impongono alcuni requisiti, e **sezioni informative**, concepite per aiutare il lettore nella comprensione, ma che non fanno parte della definizione formale. Le sezioni normative usano frequentemente locuzioni quali *deve, non deve* e *dovrebbe* rispettivamente per richiedere, proibire e suggerire aspetti dell'architettura. Per esempio, una frase tipo

L'esecuzione di un codice operativo riservato deve causare una trap

asserisce che se il programma esegue un codice operativo che non è definito, deve provocare il sollevamento di una trap e non limitarsi a ignorare l'avvenimento. Un approccio alternativo potrebbe lasciare la scelta libera, nel qual caso la frase diventerebbe

L'effetto dell'esecuzione di un codice operativo riservato è definito dall'implementazione.

Ciò significa che chi scrive il compilatore non può contare su nessun comportamento prestabilito; si lascia a ciascun produttore la libertà di fare la propria scelta. La maggior parte delle specifiche architetturali è corredata da risultati di test sperimentali che verificano la reale conformità di un'implementazione alla specifica corrispondente.

È evidente che la ragione per cui lo SPARC V9 ha un documento che ne definisce il livello ISA è fare in modo che tutti i chip SPARC V9 possano eseguire lo stesso codice. Non esiste alcun documento formale di definizione per il Pentium 4, dal momento che Intel non ha interesse ad agevolare altri produttori di chip a costruire chip del Pentium 4. Anzi, Intel è ricorsa alla giustizia nel tentativo di impedire agli altri produttori di clonare i suoi chip, ma ha perso la causa.

Un'altra proprietà importante del livello ISA è che la maggior parte dei processori è dotata di almeno due modalità d'esecuzione. La **modalità kernel** serve a eseguire il sistema operativo e permette l'esecuzione di tutte le istruzioni. La **modalità utente** ha lo scopo di eseguire i programmi applicativi e non consente l'esecuzione di certe istruzioni "delicate" (come quelle che manipolano direttamente la cache). All'interno di questo capitolo focalizziamo l'attenzione in prevalenza sulle istruzioni in modalità utente e sulle loro proprietà.

5.1.2 Modelli di memoria

Tutti i computer suddividono la memoria in celle indirizzate in modo consecutivo. Al momento la dimensione più comune delle celle è di 8 bit, ma in passato sono state usate celle di dimensioni diverse, da 1 a 60 bit (Figura 2.10). Una cella di 8 bit si chiama **byte**. La ragione per prediligere i byte è che i caratteri nella tabella ASCII occupano 7 bit, così che un carattere ASCII più un bit di parità riempiono esattamente un byte. Se in futuro lo standard UNICODE prenderà il sopravvento sul mercato, i computer potranno forse basarsi su unità consecutive di 16 bit. Dopotutto il numero 2^4 è più bello di 2^3, dal momento che 4 è una potenza di 2 mentre 3 non lo è.

In genere i byte vengono raggruppati in parole di 4 byte (32 bit) o di 8 byte (64 bit) ed esistono istruzioni apposite per manipolare intere parole. Molte architetture esigono che le parole siano allineate lungo le loro estremità e così, per esempio, una parola di 4 byte può cominciare agli indirizzi 0, 4, 8, e così via, ma non agli indirizzi 1 o 2. Allo stesso modo una parola di 8 byte può cominciare all'indirizzo 0, 8 o 16, ma non all'indirizzo 4 o 6. L'allineamento delle parole di 8 byte è illustrato nella Figura 5.2.

Spesso l'allineamento è richiesto perché in tal modo la memoria riesce a funzionare in modo più efficiente. Per esempio il Pentium 4, che effettua fetch di parole da 8 byte alla volta, usa indirizzi fisici di 36 bit, ma dispone d'indirizzamento a soli 33 bit, come mostrato nella Figura 3.45. Il Pentium 4 non potrebbe referenziare (nel senso di "riferirsi a", "puntare a") una parola non allineata in memoria neanche se lo volesse; i 3 bit meno significativi non vengono specificati esplicitamente, bensì sono posti sempre a zero, così gli indirizzi di memoria accessibili sono necessariamente multipli di 8 byte.

D'altra parte il requisito dell'allineamento può anche causare difficoltà. Nel Pentium 4 i programmi ISA possono referenziare parole che cominciano a qualsiasi indirizzo, una proprietà ereditata dall'8088 che aveva un bus dati largo 1 byte (e perciò nessun bisogno di allineare i riferimenti alla memoria lungo multipli di 8 byte). Se un programma del Pentium 4 legge una parola di 4 byte dall'indirizzo 7, l'hardware deve effettuare un accesso a memoria per recuperare i byte da 0 a 7, più un secondo accesso per recuperare i byte da 8 a 15. Infine la CPU deve estrarre i 4 byte richiesti dai 16 byte letti dalla memoria e assemblarli nel giusto ordine affinché formino una parola di 4 byte.

La capacità di leggere parole che cominciano a indirizzi arbitrari richiede nel chip funzionalità logiche supplementari, il che lo rende più grande e più costoso. Gli ingegneri di

Figura 5.2 Una parola di 8 byte in una memoria little-endian (a) allineata e (b) non allineata. Alcune macchine richiedono che le parole siano allineate.

progetto farebbero volentieri a meno di ciò, e imporrebbero che ogni programma effettuasse riferimenti alla memoria allineati. Il problema è che, ogniqualvolta gli ingegneri chiedono "a chi interessa poter eseguire codice 8088 antiquato che effettua riferimenti sbagliati in memoria?", gli addetti al marketing rispondono lapidari: "ai nostri clienti".

Gran parte dei processori a livello ISA dispone di un solo spazio lineare degli indirizzi, che si estende dall'indirizzo 0 fino a un certo massimo, generalmente 2^{32} o 2^{64} byte. Esistono tuttavia macchine che dispongono di spazi degli indirizzi separati per le istruzioni e per i dati, di modo che il fetch di un'istruzione all'indirizzo 8 proviene da un diverso spazio degli indirizzi rispetto al fetch di un dato all'indirizzo 8. Questo schema è sì più complesso di quello con spazio degli indirizzi unitario, ma presenta due vantaggi. Per prima cosa è possibile referenziare 2^{32} byte di programma e 2^{32} byte di dati usando indirizzi di soli 32 bit. In secondo luogo, poiché le scritture avvengono sempre nello spazio dei dati diviene impossibile sovrascrivere il programma accidentalmente, il che elimina una possibile sorgente di bachi di programma.

Si noti che disporre di spazi degli indirizzi separati per dati e istruzioni non è lo stesso che disporre di una cache separata di primo livello. Nel primo caso il numero totale d'indirizzi viene raddoppiato e gli accessi agli indirizzi portano a risultati differenti, a seconda che avvengano nello spazio dei dati o delle istruzioni. Nel caso della cache separata c'è un solo spazio degli indirizzi, soltanto che cache differenti ne contengono porzioni differenti.

Un ulteriore aspetto del modello di memoria a livello ISA è la semantica della memoria. È ragionevole aspettarsi che un'istruzione `LOAD`, eseguita dopo un'istruzione `STORE` e sullo stesso indirizzo, restituisca il valore appena memorizzato. Invece sappiamo dal Capitolo 4 che in molte architetture le microistruzioni sono riordinate, e questo genera il pericolo concreto che la memoria esibisca comportamenti inattesi. La faccenda si complica ulteriormente nel caso di un multiprocessore, dove ogni CPU invia un flusso di richieste di accessi in lettura e scrittura (eventualmente riordinate) alla stessa memoria condivisa.

I progettisti di sistema possono scegliere tra molti approcci risolutivi del problema. A un estremo c'è la possibilità di serializzare tutte le richieste d'accesso a memoria, così che ciascuna viene completata prima che venga emessa la successiva. Questa strategia degrada le prestazioni, ma dà luogo alla semantica di memoria più semplice in assoluto (tutte le operazioni sono eseguite esattamente nell'ordine specificato dal programma).

All'altro estremo c'è il caso in cui non si dà nessun tipo di garanzia. Per forzare un ordine sulla memoria il programma deve eseguire un'istruzione `SYNC`, che blocca l'emissione di nuove operazioni sulla memoria finché le precedenti non risultino completate. Questa scelta genera un grosso carico di lavoro supplementare per il compilatore, che deve conoscere il funzionamento della microarchitettura in dettaglio, ma assicura ai progettisti hardware la massima libertà nell'ottimizzazione dell'utilizzo della memoria.

Si danno anche modelli di memoria intermedi, in cui l'hardware blocca automaticamente l'emissione di certi accessi a memoria (per esempio quelli che coinvolgono dipendenze RAW o WAR), ma non di altri. Nonostante l'esposizione della microarchitettura a livello ISA generi tutte queste anomalie abbastanza fastidiose (quanto meno per i programmatori che scrivono il compilatore e il linguaggio assemblativo), l'abitudine è molto diffusa. Questa tendenza è dovuta alle implementazioni sottostanti, quali il riordinamento delle microistruzioni, le pipeline profonde, i livelli multipli di cache, e così via. Nel corso di questo capitolo incontreremo altri esempi di effetti così poco naturali.

5.1.3 Registri

Tutti i computer dispongono di qualche registro visibile a livello ISA. Il loro compito è il controllo dell'esecuzione del programma, il contenimento dei risultati temporanei o altro. In genere i registri visibili a livello microarchitetturale, quali il `TOS` e il `MAR` nella Figura 4.1, non sono visibili a livello ISA. Tuttavia alcuni di loro, come il program counter e il puntatore allo stack, sono visibili a entrambi i livelli. D'altro canto i registri visibili a livello ISA sono sempre visibili a livello della microarchitettura, perché è lì che sono implementati.

I registri del livello ISA possono essere divisi grosso modo in due categorie: registri specializzati e registri d'uso generale. I primi comprendono il program counter, il puntatore allo stack e altri registri dedicati a funzioni specifiche. I registri d'uso generale sono destinati invece a contenere le variabili locali più importanti e i risultati parziali del calcolo. La loro funzione principale è di consentire un accesso rapido a dati usati ricorrentemente (in pratica per evitare accessi in memoria). Le macchine RISC, dotate di CPU veloci e memorie relativamente lente, hanno in genere almeno 32 registri d'uso generale, ma nei nuovi progetti la tendenza è di incrementarne il numero.

I registri d'uso generale di alcune macchine sono del tutto simmetrici e intercambiabili. Se i registri sono tutti equivalenti il compilatore può scegliere indifferentemente se usare `R1` o `R25` per mantenere un risultato temporaneo: la scelta del registro non ha importanza.

I registri d'uso generale di altre macchine possono invece essere in qualche modo specializzati. Per esempio, il Pentium 4 ha un registro chiamato `EDX` utilizzabile come registro d'uso generale, ma che è anche destinato a ricevere la metà del prodotto in una moltiplicazione e la metà del dividendo in una divisione.

Anche quando i registri d'uso generale sono completamente intercambiabili, è usuale che i sistemi operativi o i compilatori adottino convenzioni nel modo di utilizzarli. Per esempio alcuni registri possono contenere i parametri di chiamata a una procedura e altri essere usati come registri di lavoro. Se un compilatore memorizza una variabile locale importante in `R1` e poi richiama una procedura di libreria che considera `R1` un registro di lavoro, al termine dell'esecuzione della procedura `R1` potrebbe contenere della spazzatura. Laddove esistono convenzioni su come vadano usati i registri di un sistema, è consigliabile che i compilatori e i programmatori del linguaggio assemblativo le rispettino... per evitare problemi.

Oltre ai registri del livello ISA visibili ai programmi dell'utente, esiste un numero sostanziale di registri specializzati visibile solo in modalità kernel e che controlla cache, memoria, dispositivi di I/O e altre funzionalità hardware della macchina. Possono essere impiegati solo dal sistema operativo, perciò i compilatori e gli utenti non hanno bisogno di riconoscerli.

Il **registro di flag**, detto anche `PSW` (*Program Status Word*), è una specie di ibrido tra la modalità kernel e quella utente. Questo registro di controllo contiene vari bit di natura eterogenea che sono necessari alla CPU, tra cui i più importanti sono i **codici di condizione**, che vengono impostati a ogni ciclo dell'ALU e riflettono lo stato del risultato dell'operazione più recente. Alcuni bit tipici che rappresentano codici di condizione sono:

N – posto a 1 dopo risultato negativo

Z – posto a 1 dopo risultato uguale a zero

V – posto a 1 se il risultato ha causato un overflow

C – posto a 1 se il risultato ha causato un riporto oltre l'ultimo bit più significativo

A – posto a 1 se si è verificato un riporto oltre il terzo bit (riporto ausiliario)

P – posto a 1 se il risultato è pari (parità nulla).

I codici di condizione sono importanti perché sono utilizzati dalle istruzioni di confronto e di salto condizionato. Per esempio, l'istruzione CMP sottrae due operandi e imposta il codice di condizione in base alla differenza. Se gli operandi sono uguali, la differenza è zero e il bit di codice di condizione Z viene posto a 1. Una successiva istruzione BEQ (*branch on equal*, "salta se uguali") controlla il bit Z ed effettua il salto se questo ha valore 1.

Il PSW non contiene soltanto codici di condizione, ma il resto del contenuto varia da macchina a macchina. Alcuni campi addizionali molto comuni sono la modalità di macchina (cioè, utente o kernel), i bit di traccia (usati nel debugging), il livello di priorità della CPU e lo stato di attivazione degli interrupt. Spesso il PSW è leggibile in modalità utente, ma alcuni dei suoi campi possono essere scritti solo in modalità kernel (per esempio il bit di modalità kernel/utente).

5.1.4 Istruzioni

La caratteristica principale del livello ISA è l'insieme d'istruzioni macchina che definisce, che specifica ciò che la macchina è in grado di fare. Comprende sempre le istruzioni STORE e LOAD (in forme diverse) finalizzate al trasferimento di dati dai registri alla memoria e viceversa, nonché l'istruzione MOVE per la copia di dati tra registri. Sono sempre presenti le istruzioni aritmetiche, le istruzioni booleane e quelle di confronto dei dati con eventuale salto condizionato dal risultato del confronto. Abbiamo già incontrato alcune istruzioni ISA comuni (vedi la Figura 4.11) e ne incontreremo molte di più in questo capitolo.

5.1.5 Panoramica del livello ISA del Pentium 4

In questo capitolo analizziamo estesamente tre ISA molto differenti tra loro: IA-32 di Intel, incorporato nel Pentium 4, la versione 9 dell'architettura SPARC implementata nei processori UltraSPARC, e infine l'8051. L'intento non è quello di fornire una descrizione esaustiva di questi ISA, ma piuttosto di mostrare gli aspetti importanti dell'ISA in generale e la loro variabilità da un particolare ISA all'altro. Cominciamo con il Pentium.

Il Pentium 4 è frutto dell'evoluzione di molte generazioni, e risalendo il suo albero genealogico possiamo raggiungere alcuni tra i primi processori mai costruiti, come già trattato nel Capitolo 1. Il suo ISA, oltre a mantenere completo supporto per l'esecuzione di programmi scritti per l'8086 e per l'8088 (che avevano lo stesso ISA), contiene addirittura rimembranze dell'8080, un processore a 8 bit diffusissimo negli anni '70. L'8080 fu a sua volta influenzato da vincoli di compatibilità dell'ancora precedente 8008, basato sul 4004, un chip a 4 bit in uso ai tempi in cui i dinosauri vagavano sulla Terra.

Dal punto di vista del software, sia l'8086 sia l'8088 erano macchine a 16 bit effettivi (anche se l'8088 aveva un bus dati a 8 bit). Anche il loro successore, l'80286, era una macchina a 16 bit, e presentava come principale miglioria un maggiore spazio degli indirizzi, sebbene pochi programmi se ne siano mai avvantaggiati, visto che era composto da 16.384 segmenti di 64 KB, invece che da una memoria lineare di 2^{30} byte.

L'80386 fu la prima macchina a 32 bit della famiglia Intel. Tutti i processori successivi (80486, Pentium, Pentium Pro, Pentium II, Pentium III, Pentium 4, Celeron, Xeon, Pentium M, Centrino, e così via) hanno essenzialmente la stessa architettura dell'80386, nota come **IA-32**, perciò focalizziamo la nostra attenzione su questa architettura. Gli unici cambiamenti architetturali significativi introdotti dopo l'80386 sono stati le istruzioni MMX, SSE e SSE2 nelle versioni più recenti. Si tratta d'istruzioni altamente specializzate e progettate per incrementare le prestazioni di applicazioni multimediali.

Il Pentium 4 è dotato di tre modalità operative, due delle quali lo fanno funzionare come un 8088. Nella **modalità reale** tutte le caratteristiche introdotte dopo l'8088 sono disattivate e il Pentium 4 si comporta come un semplice 8088. Se un programma fa qualcosa di sbagliato, la macchina si blocca. Se l'Intel avesse progettato gli esseri umani, li avrebbe fatti con un bit per ritornare alla modalità scimpanzé (con gran parte del cervello disattivata, privi di parola, con dimora sugli alberi, predilezione per le banane, e così via).

A un gradino più in alto c'è la **modalità virtuale 8086**, che consente di eseguire vecchi programmi dell'8088 in modo protetto. In questa modalità la macchina è controllata da un vero e proprio sistema operativo. Per eseguire un programma 8088, il sistema operativo crea un ambiente isolato speciale che si comporta come un 8088, con la sola differenza che, se il programma si blocca, la macchina non si arresta, ma passa il controllo al sistema operativo. Quando un utente di Windows apre una finestra MS-DOS, il programma che la esegue gira in modalità virtuale 8086 per proteggere lo stesso Windows da eventuali comportamenti errati dei programmi MS-DOS.

Infine abbiamo la **modalità protetta**, in cui il Pentium 4 si comporta come un Pentium 4 e non come un 8088 molto costoso. Sono previsti quattro livelli di privilegi, controllati da bit di PSW. Il livello 0, usato dal sistema operativo, corrisponde alla modalità kernel degli altri computer e ha accesso completo alla macchina. Il livello 3, usato per i programmi utente, blocca l'accesso a certe istruzioni critiche e controlla i registri per impedire a un programma utente malintenzionato di mandare in panne l'intera macchina. I livelli 1 e 2 sono usati raramente.

Il Pentium 4 dispone di uno spazio degli indirizzi enorme: la memoria è divisa in 16.384 segmenti, ciascuno dei quali va dall'indirizzo 0 all'indirizzo $2^{32} - 1$. Tuttavia la maggior parte dei sistemi operativi (compreso UNIX e tutte le versioni di Windows) supporta un solo segmento, così che la maggior parte delle applicazioni vede a tutti gli effetti solo uno spazio degli indirizzi lineare di 2^{32} byte; e talvolta parte di questo spazio è occupata dal sistema operativo. Ogni byte dello spazio degli indirizzi ha un proprio indirizzo, e le parole sono lunghe 32 bit. Le parole sono memorizzate in formato little-endian (il byte meno significativo ha indirizzo più basso).

I registri del Pentium 4 sono mostrati nella Figura 5.3. I primi quattro registri, EAX, EBX, ECX ed EDX, sono registri a 32 bit di uso più o meno generale, anche se ciascuno ha le sue peculiarità. EAX è il registro aritmetico principale; EBX è usato per contenere puntatori (a indirizzi di memoria); ECX è usato nei cicli; EDX è necessario per le moltiplicazioni e per le divisioni durante le quali, insieme a EAX, contiene prodotti e dividendi di 64 bit. Tutti questi registri sono utilizzabili come registri di 16 bit (nei 16 bit meno significativi) o di 8 bit (negli 8 bit meno significativi) e quindi agevolano la manipolazione di quantità di 16 e 8 bit rispettivamente. I registri a 32 bit furono introdotti con l'80386, come anche il prefisso E, che sta per Esteso.

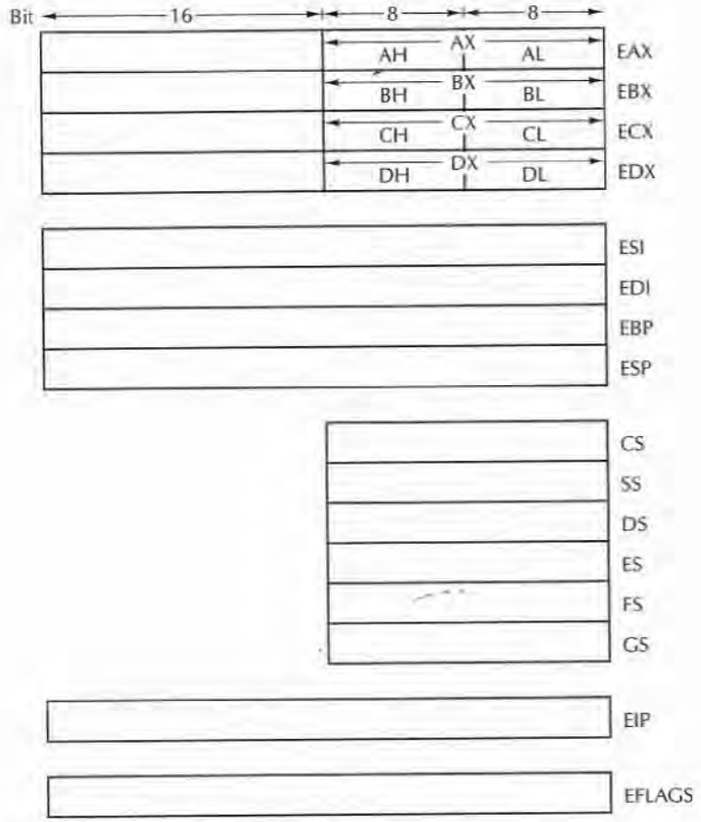


Figura 5.3 Registri principali di Pentium 4.

Anche i tre registri successivi sono in un certo senso generali, ma con maggiori peculiarità. I registri `ESI` ed `EDI` servono a contenere puntatori alla memoria, in special modo per le istruzioni hardware di manipolazione di stringhe, dove `ESI` punta alla stringa sorgente ed `EDI` alla stringa destinazione. Anche il registro `EBP` è un puntatore ed è usato generalmente per referenziare l'indirizzo base del record d'attivazione corrente, analogamente a `LV` nella IJVM. Quando un registro (come `EBP`) è usato per puntare all'indirizzo base del record d'attivazione locale viene detto per l'appunto **puntatore al record d'attivazione** (*frame pointer*). Infine `ESP` è il puntatore allo stack.

Il gruppo successivo di registri, che vanno da `CS` a `GS`, è costituito da registri di segmento. Si può dire che sono trilobiti elettronici, antichi fossili giunti fino a noi dall'epoca in cui l'8088 cercava d'indirizzare 2^{20} byte di memoria usando indirizzi da 16 bit. Basti sapere che possono essere tranquillamente ignorati quando il Pentium 4 è impostato per usare un solo spazio degli indirizzi lineare a 32 bit. Poi c'è l'`EIP` (*Extended Instruction Pointer*), che è il program counter. Infine troviamo `EFLAGS`, che è analogo a `PSW`.

5.1.6 Panoramica del livello ISA dell'UltraSPARC III

L'architettura SPARC venne introdotta per la prima volta da Sun Microsystems nel 1987, e si trattò di una delle prime architetture commerciali etichettate come RISC. Nacque come frutto dell'attività di ricerca svolta a Berkeley negli anni '80 (Patterson, 1985; Patterson e Séquin, 1982). Lo SPARC in origine aveva un'architettura a 32 bit, ma l'UltraSPARC III è una macchina a 64 bit basata sulla Versione 9, che è quella descritta in questo capitolo. Per motivi di coerenza con il resto del libro ci riferiamo qui all'UltraSPARC III anche se, a livello ISA, tutte le macchine UltraSPARC sono identiche.

La struttura della memoria dell'UltraSPARC III è chiara e semplice: la memoria indirizzabile è un vettore di 2^{64} byte. Questa memoria è così grande (18.446.744.073.709.551.616 byte) che sfortunatamente a tutt'oggi non esistono calcolatori che possano implementarla. Le implementazioni attuali hanno una limitazione sul numero d'indirizzi cui possono accedere (2^{44} byte sull'UltraSPARC III), ma tale quantità verrà incrementata nei modelli futuri. L'ordinamento usuale dei byte è il big-endian, ma lo si può convertire in little-endian agendo su di un bit apposito del registro `PSW`.

È importante che l'ISA preveda una limitazione più grande delle necessità implementative, perché in futuro quasi certamente sarà necessario incrementare la dimensione della memoria accessibile dal processore. Infatti uno dei problemi più seri che hanno dovuto affrontare le architetture di successo è stata la limitazione alla quantità di memoria imposta dal livello ISA. In informatica l'unico errore che non si può aggirare è la mancanza di bit. Un giorno i vostri nipoti vi chiederanno come potevano funzionare i computer dei vecchi tempi con indirizzi di soli 32 bit e con soli 4 GB di memoria effettiva, quando un qualsiasi videogioco avrà bisogno di almeno 8 GB solo per partire.

L'ISA dello SPARC è privo di astrusità, anche se l'organizzazione dei registri è complicata dal tentativo di rendere le chiamate di procedura più efficienti. L'esperienza dimostra che l'organizzazione dei registri causa più problemi di quanti ne risolva, ma la vecchia regola della retrocompatibilità la rende imprescindibile.

L'UltraSPARC III ha due gruppi di registri: 32 d'uso generale da 64 bit e 32 in virgola mobile. I registri d'uso generale hanno nomi che vanno da `R0` fino a `R31`, anche se possono assumere nomi diversi a seconda del contesto. La Figura 5.4 mostra i nomi alternativi e la funzione dei vari registri.

Tutti i registri d'uso generale sono di 64 bit e possono essere letti e scritti da una varietà d'istruzioni di caricamento e memorizzazione, a eccezione di `R0` contenente la costante zero. Gli utilizzi indicati nella Figura 5.4 sono in parte dovuti a convenzioni, ma anche giustificati dal modo in cui l'hardware tratta i registri. In generale è poco saggio deviare dalle modalità d'utilizzo elencate nella figura, a meno che non siate "cintura nera" di SPARC e sappiate veramente ciò che state facendo. È responsabilità del compilatore o del programmatore assicurarsi che un programma acceda ai registri in maniera corretta e che effettui l'aritmetica appropriata al caso. Per esempio, è molto facile caricare numeri in virgola mobile nei registri d'uso generale e poi effettuare su di loro l'addizione intera, il che produce un risultato che non ha alcun senso, ma che la CPU calcolerà senza batter ciglio se così istruita.

Le variabili globali sono usate per memorizzare costanti, variabili e puntatori necessari a tutte le procedure, sebbene in caso di necessità possono essere memorizzate e ricaricate all'ingresso e all'uscita di ogni procedura. I registri `Ix` e `Ox` sono usati per il passaggio di pa-

Registri	Nomi alternativi	Funzione
R0	G0	Cablato a zero: non può essere modificato
R1 – R7	G1 – G7	Contengono variabili globali
R8 – R13	O0 – O5	Contengono i parametri della procedura che viene invocata
R14	SP	Puntatore allo stack
R15	O7	Registri di lavoro
R16 – R23	L0 – L7	Contengono le variabili locali della procedura corrente
R24 – R29	I0 – I5	Contengono i parametri in ingresso
R30	FP	Puntatore alla base del record d'attivazione corrente
R31	I7	Contiene l'indirizzo di ritorno della procedura corrente

Figura 5.4 Registri d'uso generale di UltraSPARC III.

rametri a procedura per evitare accessi in memoria. Più avanti analizzeremo questo meccanismo in dettaglio.

Tre registri dedicati sono usati per scopi specifici. I registri `FP` e `SP` delimitano il record d'attivazione corrente. Il primo punta alla sua base ed è usato per indirizzare le variabili locali, al pari di `LV` nella Figura 4.10. Il secondo indica l'attuale cima dello stack e si sposta ogniqualvolta lo stack viene incrementato o decrementato, rispettivamente a seguito di operazioni di push o di pop. Invece `FP` cambia solo a seguito di chiamate di procedura e al loro ritorno. Il terzo registro specializzato è `R31`, usato per contenere l'indirizzo di ritorno delle procedure durante la loro chiamata.

In realtà l'UltraSPARC dispone di più di 32 registri d'uso generale, ma in ogni istante solo 32 di questi sono visibili al programma. Questa caratteristica, nota come **finestra di registri**, è intesa a migliorare l'efficienza delle chiamate di procedura ed è illustrata nella Figura 5.5. L'idea di base è quella di emulare uno stack, però usando i registri. In sostanza esistono diversi insiemi di registri, così come esistono diversi record d'attivazione in uno stack, e dunque sono visibili esattamente 32 registri alla volta. Il registro `CWP` (*Current Window Pointer*) tiene traccia dell'insieme di registri d'uso corrente.

L'istruzione di chiamata di procedura nasconde il vecchio insieme di registri e fornisce alla procedura chiamata un nuovo insieme, ottenuto decrementando `CWP`. C'è da dire che alcuni registri della procedura chiamante vengono riportati nel record della procedura chiamata, ottenendo così un meccanismo efficiente per il passaggio di parametri tra procedure. Questa tecnica funziona rinominando alcuni registri: dopo la chiamata di procedura i vecchi registri di output, che vanno da `R8` a `R15`, sono ancora visibili, ma costituiscono adesso i registri di input, rinominati nei registri che vanno da `R24` fino a `R31`. In ogni caso gli otto registri globali non cambiano, ovvero il loro insieme resta sempre lo stesso.

A differenza della memoria che è quasi infinita (almeno finché lo stack non ne esaurisce lo spazio), le finestre di registri disponibili si esauriscono facilmente se le procedure sono troppo annidate. In tale evenienza l'insieme meno recente è riversato in memoria per rendere disponibile un nuovo insieme. Allo stesso modo, è possibile che un insieme di registri debba essere recuperato dalla memoria dopo un gran numero di rientri di procedura. A conti

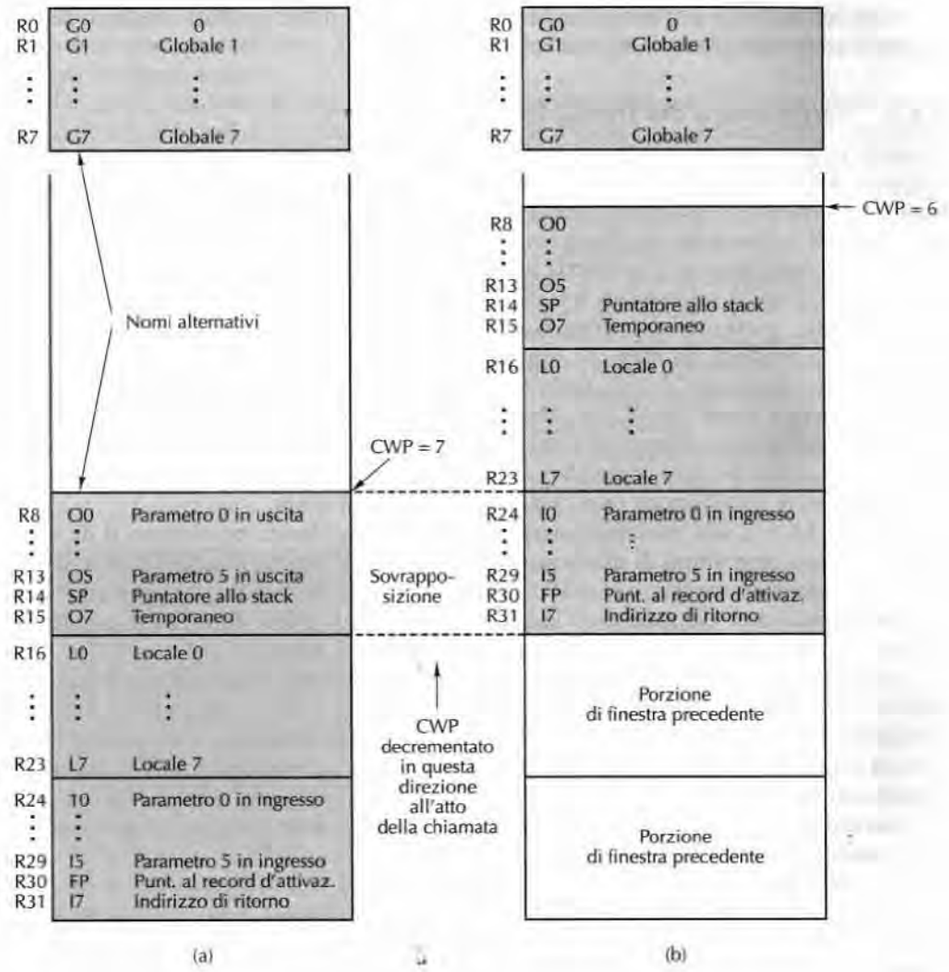


Figura 5.5 Funzionamento della finestra di registri in UltraSPARC III.

fatti, la complessità di questo meccanismo è una gran seccatura e probabilmente costituisce più un problema che una risorsa. Conviene solo nel caso in cui l'annidamento delle chiamate non sia troppo spinto.

L'UltraSPARC III dispone anche di 32 registri in virgola mobile, che possono contenere valori di 32 bit (precisione singola) o di 64 bit (precisione doppia). È anche possibile utilizzare due di questi registri per gestire valori di 128 bit (precisione quadrupla).

L'architettura UltraSPARC III è un'**architettura load/store**, ossia le sole operazioni che accedono alla memoria sono `LOAD` e `STORE`, istruzioni utili al trasferimento di dati tra i registri e la memoria. Tutti gli operandi delle istruzioni logiche e aritmetiche devono essere

contenuti nei registri o nell'istruzione stessa (non in memoria), e così tutti i risultati devono essere salvati in un registro (non in memoria).

5.1.7 Panoramica del livello ISA dell'8051

Il nostro terzo esempio è l'8051. A differenza del Pentium 4 (usato prevalentemente nei desktop e nelle *server farm*) e dell'UltraSPARC III (impiegato soprattutto nella configurazione di grandi server, specie se multiprocessore), l'8051 è usato nei sistemi integrati, quali i semafori e le radiosveglie, per il loro controllo, la gestione dei pulsanti, delle luci e delle altre parti che costituiscono l'interfaccia utente. La storia del processore è semplice: quando nel 1974 Intel lanciò sul mercato la CPU 8080 integrata in un singolo chip, fu subito un successo. Tutti i produttori lo incorporarono nei loro prodotti e quindi chiesero all'Intel se avesse potuto produrre un chip che ospitasse non solo la CPU, ma anche la memoria e i controllori dei dispositivi di I/O, così da ridurre il numero di chip da installare. Intel raccolse le richieste e lanciò l'8048, che presto diventò l'8051. Nonostante la sua veneranda età (o forse proprio a causa di questa) è ancor oggi molto usato in ragione del suo costo assai contenuto, e sappiamo che i costi sono cruciali nei sistemi integrati. In questo paragrafo forniamo una breve introduzione tecnica dell'8051 e dei suoi simili.

L'8051 ha una sola modalità e nessuna protezione hardware, poiché non si dà mai il caso che esegua programmi di utenti potenzialmente ostili. Anche il modello della memoria è estremamente semplice: c'è uno spazio degli indirizzi di 64 KB per i programmi e di altri 64 KB per i dati. Gli spazi degli indirizzi sono separati, al fine di implementare lo spazio dei programmi in una ROM e lo spazio dei dati in una RAM.

Sono altresì possibili molte altre implementazioni di memoria: nella più semplice ci sono 4 KB di ROM per i programmi e 128 KB di RAM per i dati, ed entrambe le memorie risiedono sul chip. Questa quantità di memoria è spesso sufficiente per piccole applicazioni ed è un gran risultato riuscire a contenere tutto su di un unico chip. L'8052 ha su chip il doppio di memoria: 8 KB di ROM e 256 byte di RAM. In questo schema il programma viene scritto nella ROM dell'8051 (o dell'8052) al termine della sua stesura e non può più essere modificato dall'utente.

In alternativa, è possibile procurarsi un sistema 8051 con una memoria esterna per i programmi di tipo ROM da 64 KB o di tipo EPROM, più una memoria dati RAM aggiuntiva da 64 KB; altrimenti è possibile scegliere una sola RAM esterna di 64 KB destinata a contenere allo stesso tempo dati e programmi.

L'8051 supporta anche un modello intermedio, con i primi 4 KB di memoria per i programmi e 128 byte di memoria dati residenti su chip, il resto esternamente. Per effettuare la scelta tra questi diversi modelli basta variare la tensione ad alcuni connettori del chip.

L'8051 ha un modo tutto suo di trattare i registri: la maggior parte dei programmi per 8051 è scritta come se l'8051 disponesse di 8 registri da 8 bit. È un modo comprensibile di concepire la CPU, dal momento che molte istruzioni contengono un campo di 3 bit che specifica quale registro usare. I registri sono nominati da `R0` a `R7`. In realtà esistono quattro insiemi di registri, sebbene sia visibile un solo insieme alla volta. Un campo di 2 bit del registro `PSW` determina qual è l'insieme di registri corrente. La finalità pratica del disporre di più insiemi di registri è di permettere un'elaborazione degli interrupt molto veloce. Quando viene lanciato un interrupt, il gestore degli interrupt non deve salvare tutti i registri, ma gli

basta commutare su di un altro insieme. Grazie a questa proprietà l'8051 riesce a trattare moltissimi interrupt al secondo, il che è importante per un processore progettato per sistemi integrati in tempo reale.

Un'altra peculiarità dei registri dell'8051 è che sono compresi all'interno dello spazio degli indirizzi. Il byte 0 dello spazio dei dati equivale al registro `R0` dell'insieme di registri 0. Se un'istruzione modifica `R0` e si effettua poi una lettura del byte 0, vi si trova il nuovo valore di `R0`. Allo stesso modo il byte 1 della memoria è `R1` e così via. I byte dall'8 al 15 corrispondono all'insieme di registri 1, e così di seguito fino al byte 31, che è `R7` dell'insieme di registri 3. La disposizione dei registri è mostrata nella Figura 5.6(a).

Appena sopra i banchi di quattro registri si trovano 16 byte di memoria, dall'indirizzo 32 fino al 47, indirizzabili bit per bit, dal bit 0 al bit 127. L'8051 dispone di un insieme d'istruzioni per impostare, azzerare, esaminare, fare l'AND o l'OR di bit specifici, tutti indirizzati tramite un numero compreso tra 0 e 127. Tali istruzioni operano sui byte che vanno dall'indirizzo 32 al 47 e sono utili perché i processori integrati spesso necessitano di variabili per registrare o per impostare lo stato di interruttori, lampadine e altri dispositivi di I/O. In virtù di queste locazioni speciali di memoria l'8051 può accedere a queste variabili da un bit, risparmiandosi le seguenti operazioni: il fetch di un intero byte, lo scorrimento all'in-

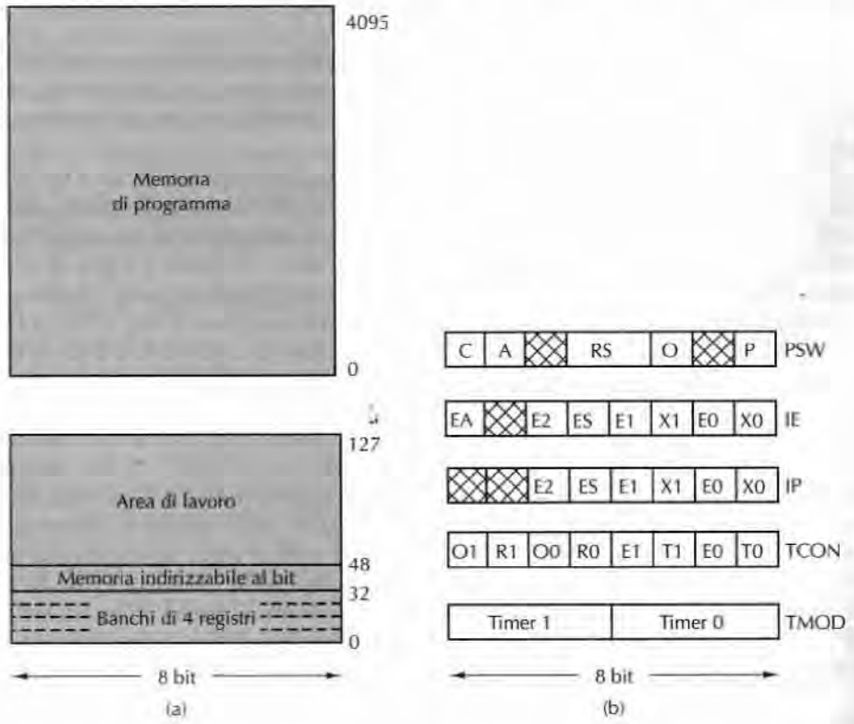


Figura 5.6 (a) Organizzazione della memoria su chip dell'8051. (b) Registri principali dell'8051.

terno del byte per accedere alla posizione giusta e infine il mascheramento dei bit indesiderati. Per essere un processore così piccolo è straordinariamente efficace nella gestione delle variabili booleane.

Oltre ai quattro insiemi di otto registri ciascuno, l'8051 dispone di un piccolo numero di registri specializzati, tra cui quelli più importanti sono indicati nella Figura 5.6(b). A partire da sinistra, PSW contiene nell'ordine: bit di riporto, bit ausiliario di riporto, l'insieme dei registri in uso, il bit di overflow e il bit di parità. Tutti questi bit sono impostati a seguito di un'istruzione aritmetica, a eccezione del bit dell'insieme dei registri. I campi retinati non sono definiti.

Il registro IE consente di attivare e disattivare gli interrupt individualmente o tutti in una volta. Se il bit EA è posto a 0, tutti gli interrupt sono disabilitati, perciò con una sola istruzione di azzeramento di questo bit è possibile disabilitare tutti gli interrupt. Se il bit è asserito, sono attivi quegli interrupt il cui bit individuale è asserito. I bit E2, E1 ed E0 abilitano o disabilitano i canali di tre timer, per cui è possibile avere tre contatori che vengono decrementati simultaneamente, ciascuno dei quali genera un interrupt quando termina il suo conto alla rovescia. Il bit ES abilita o disabilita l'interrupt del canale seriale, mentre gli altri due bit (X1 e X0) attivano o disattivano gli interrupt generati esternamente. Quando sono attivi, questi interrupt possono essere generati da alcuni dispositivi esterni collegati a due determinati connettori dell'8051. Ovviamente questi interrupt se disattivati non si verificano.

Il registro IP determina il livello di priorità di ogni interrupt; esistono due livelli: alto e basso. Un interrupt a bassa priorità può essere interrotto da uno ad alta priorità, ma non viceversa. I bit del registro IP asseriti specificano una priorità alta per gli interrupt corrispondenti, altrimenti considerati a bassa priorità.

Il registro TCON controlla i timer 0 e 1, cioè i timer principali. I bit O1 e O0 sono asseriti dall'hardware nell'istante in cui i timer corrispondenti vanno in overflow. I bit R1 e R0 sono i bit di controllo dell'esecuzione e consentono ai programmi di far partire o di fermare i timer (quindi, via software). I bit rimanenti servono a impostare il tipo di commutazione dei timer, scegliendo tra commutazione sul fronte e commutazione a livello.

L'ultimo registro, TMOD, specifica la modalità di ciascun timer (8 bit, 13 bit o 16 bit), se debba cioè comportarsi come un vero timer o solo come un contatore, e il grado in cui i segnali hardware possono controllare i timer. Esistono altri registri (non mostrati) che hanno a che fare con la gestione dell'alimentazione e il controllo della porta seriale.

Tutti i registri specializzati fin qui menzionati e pochi altri, (come l'accumulatore e le porte di I/O) si trovano nelle locazioni di memoria da 128 a 255. Il loro accesso avviene come per ogni altra locazione di memoria, allo stesso modo dei registri da R0 a R7. Per esempio l'**accumulatore**, usato dalla maggior parte delle istruzioni aritmetiche, si trova all'indirizzo 240. Nell'8052, in cui la memoria reale si trova nell'intervallo da 128 a 255, i registri specializzati ricoprono interamente lo spazio di memoria. Quando l'8052 è usato con indirizzamento diretto, i registri speciali sono indirizzati direttamente, mentre la memoria è referenziata indirettamente (attraverso puntatori nei registri) e si trova nella RAM.

5.2 Tipi di dati

Tutti i computer hanno bisogno di dati. In effetti lo scopo di molti sistemi di calcolo è proprio quello di elaborare dati finanziari, commerciali, scientifici, ingegneristici o di altro tipo. All'interno del computer, i dati devono essere rappresentati in una forma specifica. A livello ISA sono disponibili una varietà di tipi di dati diversi che esamineremo in seguito.

Una delle questioni chiave è la presenza o meno di supporto hardware per un particolare tipo di dati. Supporto hardware vuol dire che una o più istruzioni si aspettano i dati in un formato particolare e l'utente non è libero di scegliere un formato differente. Per esempio i contabili hanno la singolare abitudine di scrivere i numeri negativi con il segno meno alla destra del numero invece che alla sua sinistra, dove lo mettono gli informatici. Immaginate che il responsabile del centro di calcolo di uno studio di contabilità decida di fare una buona impressione sul suo direttore modificando i numeri di tutti i computer di modo che usino il bit meno significativo (invece del più significativo) come bit di segno. Ciò produrrebbe sicuramente un grande effetto sul direttore, infatti tutto il software inizierebbe a fare degli errori assurdi. L'hardware si aspetta un certo formato d'interi e non funziona correttamente se gli si passa altro.

Considerate ora un altro studio di contabilità, questa volta incaricato dallo Stato di valutare il debito pubblico. Non basterebbe certo l'aritmetica a 32 bit perché i numeri coinvolti sono molto più grandi di 2^{32} (pari a 4 milioni circa). Una possibile soluzione potrebbe essere l'utilizzo di due interi di 32 bit per rappresentare ciascun numero, il che fa 64 bit in tutto. Se la macchina non supporta questo tipo di numeri in **precisione doppia**, si rende necessaria la gestione software di tutta la loro aritmetica, e in tal caso l'ordine delle due parti costituenti il numero non conta, visto che l'hardware non ne viene coinvolto. È questo un esempio di tipo di dati non supportato dall'hardware, per cui quindi non è richiesta una rappresentazione hardware. Nei prossimi paragrafi ci concentreremo sui tipi di dati supportati dall'hardware, per i quali sono quindi richiesti formati specifici.

5.2.1 Tipi di dati numerici

I tipi di dati possono essere divisi in due categorie: numerici e non. Il primo tipo di dati numerici è costituito dagli interi. Ci sono interi di lunghezze diverse, in genere di 8, 16, 32 e 64 bit. Gli interi servono a contare oggetti (per esempio il numero di cacciaviti nel magazzino di un ferramenta), per identificare oggetti (per esempio il numero di conto corrente bancario) e per molto altro. La maggior parte dei computer moderni memorizza gli interi nella notazione binaria in complemento a due, pur se in passato sono stati usati anche altri sistemi. I numeri binari sono analizzati in dettaglio nell'Appendice A.

Alcuni computer supportano sia gli interi senza segno sia quelli con segno. Nel primo caso non c'è alcun bit di segno e tutti i bit rappresentano dati. Questo tipo di dati presenta il vantaggio di disporre di un bit in più e così una parola di 32 bit può contenere un intero senza segno di valore compreso tra 0 e $2^{32}-1$, estremi inclusi. D'altra parte un intero con segno di 32 bit rappresentato in complemento a due può gestire numeri minori o uguali a $2^{31}-1$ ma, naturalmente, può gestire anche numeri negativi.

Per rappresentare i numeri che non possono essere espressi con gli interi (per esempio 3,5) si usano i numeri in virgola mobile, trattati nell'Appendice B. Questi sono lunghi 32, 64 o a volte 128 bit. Quasi tutti i computer dispongono d'istruzioni per svolgere

aritmetica in virgola mobile, e la maggioranza di questi ha registri separati per contenere operandi interi e operandi in virgola mobile.

Alcuni linguaggi di programmazione, in particolare il COBOL, mettono a disposizione un tipo di dati per i numeri decimali. Le macchine che vogliono favorire l'uso del COBOL spesso supportano i numeri decimali nell'hardware, codificando ogni cifra decimale con 4 bit e quindi impacchettando due cifre decimali in un byte (*Binary Coded Decimal*, "decimali in codifica binaria" o BCD). Purtroppo l'aritmetica dei numeri decimali impacchettati non funziona molto bene, perciò si rendono necessarie delle istruzioni di correzione-decimale-aritmetica. Queste istruzioni hanno bisogno di conoscere il riporto oltre il terzo bit, ed è per questo che i codici di condizione spesso contengono anche un bit di riporto ausiliario. A margine di ciò, l'infame baco del millennio, detto anche *Y2K* (da *Year 2000*), è stato causato dalla decisione dei programmatori COBOL di usare due sole cifre decimali per rappresentare l'anno, perché più economico rispetto all'uso di un numero binario di 16 bit. Questione di ottimizzazione.

5.2.2 Tipi di dati non numerici

Sebbene i primi computer si siano guadagnati da vivere macinando numeri, i computer moderni sono spesso impiegati per svolgere applicazioni non numeriche, quali spedizione di e-mail, navigazione su Internet, fotografia digitale, creazione di contenuti multimediali e loro riproduzione. Queste applicazioni richiedono altri tipi di dati spesso supportati dalle istruzioni del livello ISA. Ovviamente i caratteri sono un tipo di dati importante in questo contesto, ma non tutti i computer ne forniscono il supporto hardware. I codici carattere più comuni sono ASCII e UNICODE, che definiscono rispettivamente caratteri di 7 e di 16 bit. Per una loro definizione dettagliata si veda il Capitolo 2.

Non è raro che il livello ISA comprenda istruzioni speciali per la gestione di stringhe di caratteri. Le stringhe sono spesso delimitate da un carattere speciale di fine stringa, oppure dotate di un campo lunghezza che può essere usato per ricavare la terminazione della stringa.

Anche i valori booleani sono importanti. Un valore booleano può assumere solo uno dei seguenti valori: vero o falso. In teoria un bit solo basta a rappresentare un dato booleano, associando 0 a falso e 1 a vero (o viceversa). In pratica vengono usati comunque interi byte per rappresentare questi dati, dal momento che i bit individuali non sono indirizzabili direttamente e sono perciò difficili da accedere. Comunemente si adotta la convenzione secondo cui 0 vuol dire falso e tutto il resto rappresenta il valore vero.

L'unica situazione in cui un valore booleano è rappresentato da un solo bit è all'interno di array di booleani, laddove una parola di 32 bit può contenere 32 valori binari. Una struttura di dati siffatta è detta **bit map** ("mappa di bit") e la si ritrova in molti contesti. Per esempio si può usare una bit map per tener traccia dei blocchi liberi di un disco. Se un disco ha n blocchi, la bit map è lunga n bit.

L'ultimo tipo di dati che consideriamo è il tipo puntatore, nient'altro che un indirizzo di macchina. Abbiamo già incontrato i puntatori più volte. Nelle macchine Mic-x i registri `SP`, `PC`, `LV` e `CPP` sono tutti esempi di puntatori. Un'operazione molto comune su tutte le macchine consiste nell'usare i puntatori per accedere a variabili che si trovano a una distanza prefissata da loro, che è precisamente il modo in cui funziona `ILOAD`.

Tipo	1 bit	8 bit	16 bit	32 bit	64 bit	128 bit
Bit						
Interi con segno		x	x	x		
Interi senza segno		x	x	x		
Interi decimali in codifica binaria (BCD)		x				
Numeri in virgola mobile				x	x	

Figura 5.7 Tipi di dati numerici di Pentium 4 (indicati dalla crocetta x).

5.2.3 Tipi di dati del Pentium 4

Il Pentium 4 supporta gli interi con segno in complemento a due, gli interi senza segno, i numeri decimali in codifica binaria e i numeri in virgola mobile nel formato IEEE 754 (come indicato nella Figura 5.7). A causa delle sue umili origini di macchina a 8/16 bit, il Pentium 4 gestisce anche gli interi di quelle lunghezze, e mette a disposizione numerose istruzioni per gestire la loro aritmetica, le operazioni booleane e i confronti su di loro. Gli operandi non devono essere necessariamente allineati in memoria, ma se lo sono, cioè se gli indirizzi delle parole sono multipli di 4 byte, le prestazioni ne risultano incrementate.

Il Pentium 4 è ben fornito anche nella manipolazione di caratteri ASCII di 8 bit: ci sono istruzioni speciali per la copia e la ricerca di stringhe di caratteri. Queste istruzioni si applicano sia alle stringhe di lunghezza nota, sia alle stringhe con carattere di fine stringa e sono usate molto di frequente dalle librerie che manipolano le stringhe.

5.2.4 Tipi di dati dell'UltraSPARC III

L'UltraSPARC III supporta un ampio spettro di formati di dati, come evidenziato nella Figura 5.8. Per quanto riguarda i soli interi, può gestire operandi di 8, 16, 32 e 64 bit, con o senza segno. Gli interi con segno usano il formato in complemento a due. Sono disponibili operandi in virgola mobile di 32, 64 e 128 bit, dove i primi due rispettano lo standard IEEE 754. I numeri decimali in rappresentazione binaria non sono supportati. Tutti gli operandi devono essere allineati in memoria.

L'UltraSPARC III è altamente orientato all'uso dei registri e quasi tutte le istruzioni operano su registri di 64 bit. Non ci sono istruzioni hardware speciali per il supporto di tipi di dati carattere e stringa. La loro manipolazione avviene totalmente via software.

Tipo	1 bit	8 bit	16 bit	32 bit	64 bit	128 bit
Bit						
Interi con segno		x	x	x	x	
Interi senza segno		x	x	x	x	
Interi decimali in codifica binaria (BCD)						
Numeri in virgola mobile				x	x	x

Figura 5.8 Tipi di dati numerici di UltraSPARC III (indicati dalla crocetta x).

Tipo	1 bit	8 bit	16 bit	32 bit	64 bit	128 bit
Bit	×					
Interi con segno		×				
Interi senza segno						
Interi decimali in codifica binaria (BCD)						
Numeri in virgola mobile						

Figura 5.9 Tipi di dati numerici di 8051 (indicati dalla crocetta ×).

5.2.5 Tipi di dati dell'8051

L'8051 dispone di una varietà di tipi di dati molto limitata. Tutti i registri sono lunghi 8 bit, con una sola eccezione, e così anche gli interi sono di 8 bit, così come i caratteri. In pratica il solo tipo di dati le cui operazioni aritmetiche sono realmente supportate dall'hardware è il byte di 8 bit, come si evince dalla Figura 5.9.

L'8051 ha anche un altro tipo di dati supportato dall'hardware, ma non usato nell'aritmetica, cioè il bit. I 16 byte di memoria che cominciano all'indirizzo 32 sono orientati al bit, nel senso che ogni bit può essere indirizzato individualmente tramite di offset da 0 a 127. Il bit 0 è il bit meno significativo del byte 32, il bit 1 è il suo vicino e così via. Ci sono istruzioni per asserire, azzerare, per fare l'AND, l'OR e il complemento di singoli bit, per trasferirli o esaminarne il contenuto. Nei sistemi integrati i bit sono utilizzati per memorizzare lo stato degli interruttori, delle luci, e così via, perciò la capacità di manipolarli direttamente si rivela molto utile.

5.3 Formati d'istruzione

Un'istruzione consiste in un opcode (codice operativo), di solito corredato da altre informazioni quali la provenienza degli operandi e la destinazione dei risultati. L'argomento generale che tratta della provenienza degli operandi (cioè il loro indirizzo) è detto **indirizzamento** e verrà illustrato in dettaglio nel prosieguo di questo paragrafo.

La Figura 5.10 mostra i diversi formati delle istruzioni di livello 2. Ogni istruzione è associata a un opcode che ne specifica il comportamento, e può contenere nessuno, uno, due o tre indirizzi.

Alcune macchine hanno tutte le istruzioni della stessa lunghezza, altre dispongono d'istruzioni di lunghezza diversa. Le istruzioni possono essere più corte, altrettanto lunghe o più lunghe della dimensione di parola. La scelta di avere tutte le istruzioni della stessa lunghezza semplifica la loro decodifica, ma spesso implica uno spreco di spazio, visto che tutte le istruzioni devono essere lunghe quanto la più lunga. La Figura 5.11 illustra alcune delle relazioni che possono intercorrere tra la lunghezza delle istruzioni e la dimensione di parola.

Figura 5.10 Quattro formati d'istruzioni: (a) Istruzione senza indirizzi. (b) Istruzione a un indirizzo. (c) Istruzione a due indirizzi. (d) Istruzione a tre indirizzi.

5.3.1 Criteri progettuali per i formati d'istruzioni

La scelta dei formati d'istruzione prevede che i progettisti di computer di tengano conto di molti fattori. Si tratta di una decisione difficile, da non sottovalutare, e che va presa molto presto nel processo di progettazione. Se poi il computer ha successo commerciale è possibile che il suo insieme d'istruzioni sopravviva per altri vent'anni o più. Una capacità molto preziosa è quella di saper aggiungere nuove istruzioni all'insieme per sfruttare le opportunità che si possono presentare nel corso degli anni, sempre che l'architettura, e l'azienda che l'ha progettata, sopravvivano abbastanza a lungo perché il progetto si affermi.

L'efficienza di un determinato ISA dipende fortemente dalla tecnologia impiegata nell'implementazione del computer. La tecnologia cambia molto velocemente con il passare degli anni e alcune scelte fatte per l'ISA possono rivelarsi (con il senno di vent'anni dopo) infelici. Per esempio un progetto basato sullo stack (come quello dell'IJVM) è molto valido se gli accessi in memoria sono veloci, altrimenti la strada da percorrere è quella di avere molti registri (come fa l'UltraSPARC). Se pensate che sia questa una scelta facile da prendere vi invito a procurarvi un foglio e a scriverci le vostre previsioni riguardo (1) la velocità di una CPU tipica e (2) il tempo di accesso a una comune memoria RAM reperibili nei computer tra vent'anni. Piegate il foglio, conservatelo e rileggetelo al momento opportuno. Altrimenti pubblicate oggi stesso le vostre previsioni direttamente su Internet.

Naturalmente anche i progettisti più lungimiranti a volte sbagliano. E anche se non sbagliassero, devono spesso tener in considerazione il breve termine: se il loro progetto ele-

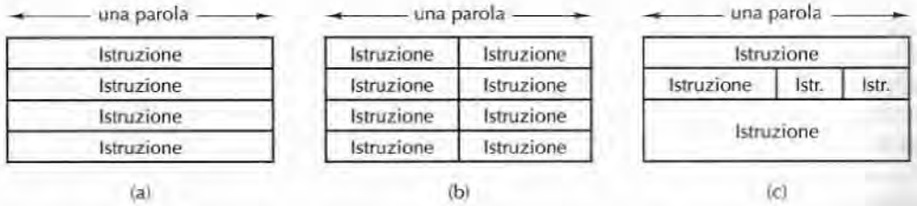


Figura 5.11 Possibili relazioni tra lunghezza delle istruzioni e dimensione di parola.

gante di ISA costasse anche solo poco più dello scadente progetto concorrente, l'azienda potrebbe non sopravvivere abbastanza perché il mondo riesca ad apprezzare l'eleganza del loro progetto.

A parità di progetto, le istruzioni corte sono preferibili a quelle lunghe. Un programma fatto di n istruzioni di 16 bit occupa metà spazio di memoria rispetto a un programma di n istruzioni di 32 bit. Questo fattore potrebbe risultare sempre meno importante in futuro, a giudicare dalla continua diminuzione del prezzo della memoria, se non fosse che le dimensioni del software metastatizzano con una velocità superiore rispetto alla riduzione del prezzo della stessa.

Minimizzare la dimensione delle istruzioni potrebbe renderle per giunta più difficili da decodificare o da sovrapporre. In conseguenza di ciò, il criterio di ottenere istruzioni di dimensione minima deve essere valutato in base al tempo richiesto per la decodifica e l'esecuzione delle istruzioni.

Un'altra ragione importante per minimizzare la dimensione delle istruzioni, e che assumerà maggior rilievo al crescere delle prestazioni dei processori, è legata all'ampiezza (o larghezza) di banda della memoria (il numero di bit che può trasferire in un secondo). Nell'ultimo decennio abbiamo assistito a una crescita impressionante della velocità dei processori, che non è stata affiancata da un incremento altrettanto significativo dell'ampiezza di banda della memoria. E così le deficienze dei sistemi di memoria, che non sono in grado di trasferire istruzioni e operandi con la stessa celerità con cui vengono processati dalla CPU, diventano sempre più un freno per i processori. La larghezza di banda delle memorie dipende dal loro contenuto tecnologico e dalla qualità della ingegnerizzazione del progetto.

Non solo la memoria principale costituisce un collo di bottiglia, ma anche le cache: se la larghezza di banda di un'istruzione in cache è di t bps e se la lunghezza media di un'istruzione è di r bit, la cache può distribuire al massimo t/r istruzioni al secondo. Si noti che questa quantità è un *limite superiore* al tasso di prestazione del processore, anche se si stanno facendo molti sforzi di ricerca per oltrepassare questa barriera apparentemente invalicabile. Ovviamente il ritmo con cui vengono eseguite le istruzioni (cioè la velocità del processore) è legato alla lunghezza delle istruzioni: istruzioni più corte significano un processore più veloce. Dal momento che molti processori moderni sono in grado di eseguire più istruzioni nello stesso ciclo di clock, si impone la necessità di effettuare fetch multipli a ogni ciclo. Questo aspetto della cache delle istruzioni rende la dimensione delle istruzioni un criterio progettuale importante, le cui conseguenze hanno una forte influenza sulle prestazioni.

Il secondo criterio progettuale è prevedere nel formato delle istruzioni spazio sufficiente a esprimere tutte le operazioni volute. Progettare una macchina con 2^n operazioni, ognuna delle quali esprimibile con meno di n bit, è impossibile. Non ci sarebbe neanche lo spazio sufficiente nel codice operativo per indicare di che istruzione si tratti. La storia è piena di esempi di progetti folli in cui non sono stati previsti abbastanza codici operativi in eccedenza per fronteggiare aggiunte successive all'insieme d'istruzioni.

Un terzo criterio concerne il numero di bit in un campo degli indirizzi. Si consideri il progetto di una macchina con caratteri di 8 bit e con una memoria principale sufficiente a contenere 2^{32} caratteri. I progettisti potrebbero scegliere d'indirizzare unità di 8, 16, 24 o 32 bit, ma anche di altre dimensioni.

Immaginate che cosa succederebbe se il gruppo di progettazione si dividesse in due fazioni agguerrite, una a sostegno del byte come unità base della memoria, l'altra favorevole

ad avere parole di 32 bit. I primi proporrebbero una memoria di 2^{32} byte, numerati da 0 a 4.294.967.295, mentre i secondi proporrebbero una memoria di 2^{30} parole, numerate da 0 a 1.073.741.823.

Il primo gruppo porrebbe l'accento sul fatto che, nell'organizzazione con parole di 32 bit, per confrontare due caratteri un programma non solo dovrebbe fare il fetch delle intere parole contenenti i caratteri, ma dovrebbe poi estrarre i caratteri dalle parole al fine di confrontarli. Così facendo si richiederebbero istruzioni aggiuntive e si sprecherebbe quindi dello spazio. L'organizzazione a 8 bit, d'altro canto, fornirebbe un indirizzo per ogni carattere, rendendo il confronto molto più semplice.

I sostenitori della parola di 32 bit replicherebbero indicando il fatto che la loro proposta comporterebbe 2^{30} indirizzi diversi, raggiungibili con soli 30 bit invece che con i 32 richiesti dall'altra proposta per indirizzare la stessa memoria. Indirizzi più corti consentono istruzioni più corte, che occuperebbero perciò meno spazio e richiederebbero meno tempo in fase di fetch. In alternativa si potrebbe mantenere in uso gli indirizzi di 32 bit e referenziare così una memoria di 16 GB invece di miseri 4.

Questo esempio dimostra che una maggiore risoluzione nell'accesso alla memoria si paga al prezzo d'indirizzi più lunghi e conseguentemente d'istruzioni più lunghe. Il massimo di risoluzione si ha quando l'organizzazione di memoria è tale per cui ogni bit è indirizzabile direttamente (come, per esempio, nel Burroughs B1700). All'altro estremo ci sono le memorie costituite da parole molto lunghe (per esempio, la serie Cyber della CDC aveva parole di 60 bit).

I computer moderni hanno raggiunto un compromesso che, in un certo senso, racchiude il peggio di entrambe le scelte. Da un lato si usano tanti bit quanti sono necessari per indirizzare individualmente tutti i byte, dall'altra spesso si accede alla memoria per leggere una, due, o addirittura quattro parole alla volta. Tanto per fare un esempio, la lettura di un byte di memoria comporta nell'UltraSPARC III la lettura di almeno 16 byte (si veda la Figura 3.48) e probabilmente di un'intera linea di cache di 64 byte.

5.3.2 Codice operativo espandibile

Nel paragrafo precedente abbiamo visto come una lunghezza degli indirizzi contenuta e una buona risoluzione di memoria si ottengano l'una a discapito dell'altra. In questo paragrafo esaminiamo altri compromessi che coinvolgono i codici operativi e gli indirizzi. Si consideri un'istruzione lunga $(n + k)$ bit, composta da un opcode di k bit e da un solo indirizzo di n bit. Un'istruzione siffatta consente 2^k operazioni diverse e permette d'indirizzare 2^n celle di memoria. In alternativa gli stessi $(n + k)$ bit potrebbero essere spezzati in $(k - 1)$ bit di opcode e $(n + 1)$ bit d'indirizzo, dimezzando il numero d'istruzioni, ma al contempo raddoppiando la dimensione della memoria raggiungibile, oppure raggiungendo la stessa quantità di memoria ma con risoluzione doppia. Viceversa la scelta di avere opcode di $(k + 1)$ bit e indirizzi di $(n - 1)$ bit rende di più in termini di operazioni, ma al prezzo di un minor numero di celle indirizzabili, o di una peggiore risoluzione nell'accesso alla stessa quantità di memoria. Sulla falsariga degli esempi appena esposti, si possono costruire compromessi molto più sofisticati tra lunghezza degli opcode e lunghezza degli indirizzi.

Introduciamo ora il concetto di **codice operativo espandibile**, che descriviamo mediante un semplice esempio. Si consideri una macchina in cui le istruzioni sono lunghe 16 bit e

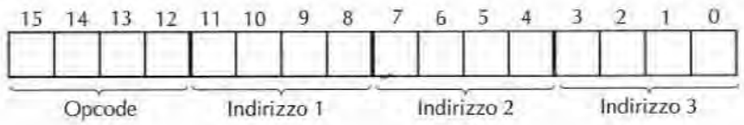


Figura 5.12 Un'istruzione composta da un opcode di 4 bit e da tre campi d'indirizzo di 4 bit.

gli indirizzi 4 bit, come nella Figura 5.12. Questa situazione potrebbe essere ragionevole per una macchina che ha 16 registri (dunque indirizzabili con 4 bit), sufficienti per lo svolgimento di tutte le operazioni aritmetiche. Un'architettura possibile sarebbe quella di comporre ogni istruzione con 4 bit di opcode e tre indirizzi, per un totale di 16 bit.

Tuttavia, se i progettisti avessero bisogno di 15 istruzioni con tre indirizzi, 14 con due indirizzi, 31 con un indirizzo e 16 senza alcun indirizzo, potrebbero usare gli opcode da 0 a 14 per le istruzioni da tre indirizzi e interpretare l'opcode 15 diversamente (Figura 5.13).

L'opcode 15 indica che il codice operativo è contenuto nei bit da 8 a 15 invece che da 12 a 15. I bit da 0 a 3 e quelli da 4 a 7 formano ancora due indirizzi, come di consueto. Le 14 istruzioni con due indirizzi hanno tutte i 4 bit più significativi posti a 1111, mentre i bit di posto da 8 a 11 progrediscono da 0000 a 1101. Le istruzioni con bit più significativi pari a 1111 e bit da 8 a 11 pari a 1110 oppure a 1111 vengono trattate diversamente, come se il loro opcode fosse contenuto nei bit da 4 a 15. Si dispone così di 32 nuovi opcode. Poiché ne sono richiesti solo 31, l'opcode 111111111111 è interpretato come se il vero opcode fosse costituito dai bit da 0 a 15, il che fornisce altre 16 istruzioni senza indirizzi.

Nel corso della spiegazione abbiamo visto crescere la dimensione dell'opcode sempre più: da 4 bit per le istruzioni con tre indirizzi, a 8 bit per le istruzioni con due indirizzi, a 12 bit per le istruzioni con un indirizzo, per finire a 16 bit per le istruzioni senza indirizzi.

L'idea di codice operativo espandibile evidenzia il compromesso che sussiste tra lo spazio riservato all'opcode e lo spazio per le altre informazioni. Nella pratica i codici operativi espandibili non sono così regolari come nel nostro esempio. La capacità di usare dimensioni di opcode variabili può venire sfruttata in due modi diversi: in primo luogo si può mantenere costante la dimensione delle istruzioni, assegnando gli opcode più corti alle istruzioni che hanno bisogno di più bit per specificare informazioni di altra natura; in secondo luogo, è possibile minimizzare la lunghezza media delle istruzioni, scegliendo gli opcode più corti possibile per le istruzioni comuni e lasciando i più lunghi per le istruzioni d'uso più raro.

Se si spinge l'idea di opcode di lunghezza variabile fino alle sue estreme conseguenze, è possibile minimizzare la lunghezza media delle istruzioni codificandone ciascuna con il minimo numero di bit necessari. Sfortunatamente ciò comporterebbe istruzioni di lunghezze diverse e non allineate nemmeno al byte. Sebbene siano esistiti ISA con questa proprietà (per esempio lo sventurato Intel 432), l'importanza dell'allineamento è così rilevante per la decodifica rapida delle istruzioni che un tale grado di ottimizzazione è quasi certamente controproducente. Nondimeno viene spesso impiegato a livello dei byte.

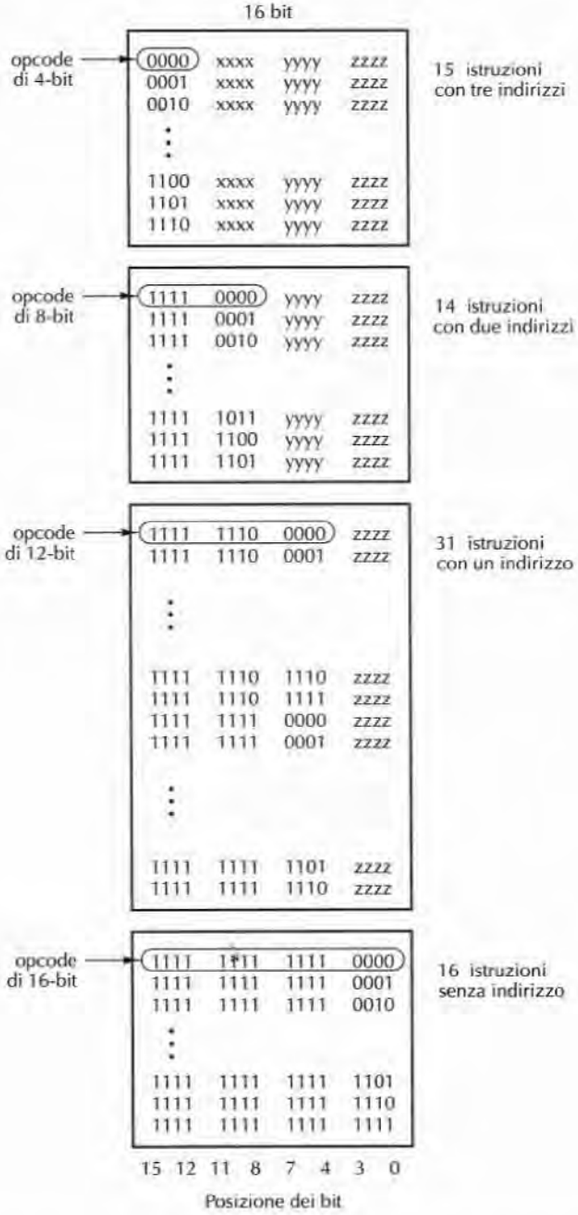


Figura 5.13 Un opcode espandibile che consente di avere 15 istruzioni con tre indirizzi, 14 con due indirizzi, 31 con un indirizzo e 16 senza indirizzo. I campi contrassegnati con *xxxx*, *yyyy* e *zzzz* sono di 4 bit.

5.3.3 Formati delle istruzioni di Pentium 4

I formati delle istruzioni del Pentium 4 sono molto complessi e irregolari, con fino a sei campi di lunghezza variabile, cinque dei quali sono opzionali. Il loro schema generale è illustrato nella Figura 5.14. Lo stato delle cose è dovuto al fatto che l'architettura si è evoluta per molte generazioni e fin dagli inizi ha incluso alcune scelte inadeguate che non sono più state messe in discussione, nel nome della retrocompatibilità. Come regola generale per le istruzioni con due operandi, se un operando si trova in memoria, l'altro non vi si può trovare, perciò esistono istruzioni per fare la somma di due registri, di un registro e una locazione di memoria, ma non di due locazioni di memoria.

Nelle prime architetture Intel tutti i codici operativi erano lunghi un byte, sebbene venisse usato largamente il concetto di byte prefisso per modificare alcune istruzioni. Un **byte prefisso** è un codice operativo supplementare preposto all'inizio di un'istruzione per cambiarne il comportamento. L'istruzione `WIDE` dell'IJVM è un esempio di byte prefisso. A un certo stadio dell'evoluzione dei suoi processori, Intel esaurì malauguratamente gli opcode disponibili, così l'opcode 0xFF fu scelto per designare un **codice di escape**, a indicare la presenza di un secondo byte per la specifica dell'istruzione.

I singoli bit degli opcode del Pentium 4 danno ben poche informazioni riguardo le istruzioni. L'unica struttura riconoscibile nel campo opcode di alcune istruzioni è l'uso del bit meno significativo per distinguere tra istruzioni di un byte o di una parola, e l'uso del bit adiacente per stabilire se l'eventuale indirizzo di memoria è da considerarsi come sorgente o come destinazione dell'operazione. Perciò l'opcode va interamente decodificato per determinare quale classe di operazioni eseguire e, di conseguenza, quanto è lunga l'istruzione corrispondente. Questo si traduce nella difficoltà di ottenere implementazioni con prestazioni elevate, dal momento che si richiede una decodifica impegnativa prima di poter stabilire dove comincia l'istruzione successiva.

In gran parte delle istruzioni che prevedono un operando in memoria, dopo il byte di opcode si trova un byte (di modo) contenente tutte le informazioni relative all'operando; è composto da un campo `MOD` da 2 bit e da due campi di 3 bit, `REG` e `R/M`. Talvolta i primi 3 bit di questo byte sono usati come estensione dell'opcode, dando vita a un opcode di 11 bit in totale. A ogni modo, il campo `MOD` di due bit consente solo quattro modalità d'indi-

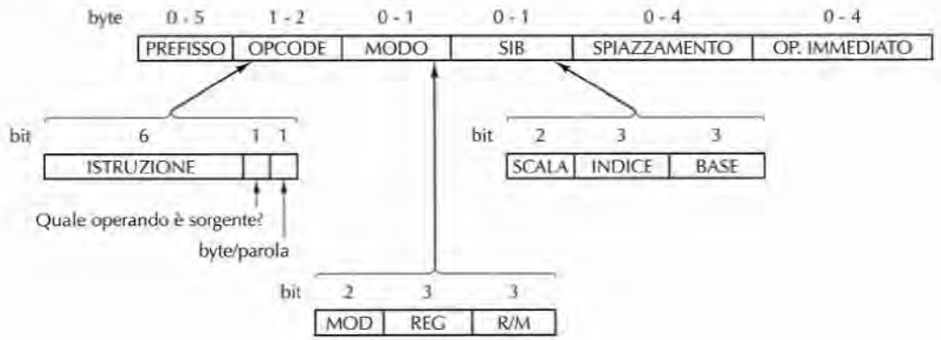


Figura 5.14 Formati d'istruzione di Pentium 4.

rizzamento degli operandi e almeno un operando è vincolato a trovarsi nei registri; secondo logica ciascuno dei registri `EAX`, `EBX`, `ECX`, `EDX`, `ESI`, `EDI`, `EBP`, `ESP` dovrebbe poter fungere da operando, ma le regole di codifica proibiscono alcune combinazioni e le riservano a casi particolari. Alcune modalità richiedono un bit supplementare, detto **SIB** (da *Scala*, *Indice* e *Base*). Nel complesso questo schema non è certo ideale, bensì un compromesso tra esigenze contrastanti, da un lato la retrocompatibilità e dall'altro la voglia di aggiungere nuove funzionalità originariamente non previste.

Inoltre, alcune istruzioni sono dotate di 1, 2 o 4 byte addizionali che specificano l'indirizzo di memoria (lo spiazzamento) e addirittura di altri 1, 2 o 4 byte atti a contenere una costante (operando immediato).

5.3.4 Formati delle istruzioni di UltraSPARC III

L'ISA dell'UltraSPARC III contiene esclusivamente istruzioni di 32 bit allineate in memoria. In genere le istruzioni sono molto semplici e ciascuna specifica una singola azione. Una tipica istruzione aritmetica specifica due indirizzi che forniscono gli operandi sorgente e un solo registro come destinazione. Una variante dell'istruzione consente di specificare una costante con segno di 13 bit al posto di uno dei registri. Nel caso di una `LOAD` viene sommato il contenuto di due registri (oppure di un registro a una costante di 13 bit) per individuare l'indirizzo di memoria cui accedere in lettura. I dati vengono quindi scritti nell'ultimo registro specificato.

Lo SPARC originale disponeva di ben pochi formati d'istruzione, cioè quelli elencati nella Figura 5.15. Con il passare del tempo sono stati aggiunti nuovi formati e, alla stesura di questo libro, il conteggio ha raggiunto quota 31 ed è in aumento (fra non molto qualche azienda la pubblicizzerà come "la macchina RISC più complessa del mondo"!). La maggior parte delle nuove varianti è derivata dall'eliminazione di pochi bit da qualche campo. Per esempio, i salti in origine usavano il formato 3 con uno spiazzamento di 22 bit ma, all'aggiunta delle istruzioni di salto predetto, 3 bit dei 22 sono stati rimossi e utilizzati come segue: uno per la previsione (salto intrapreso/non intrapreso), e gli altri per specificare quale condizione usare. Con ciò lo spiazzamento si è ridotto a 19 bit. Un altro esempio: ci sono molte istruzioni per la conversione tra tipi di dati (da interi a numeri in virgola mobile, e

Figura 5.15 Formati d'istruzione di SPARC originale.

così via) e molte di queste usano varianti del formato 1b, laddove il campo della costante immediata è spezzato in un campo di 5 bit per il registro sorgente e uno di 8 bit a disposizione dell'opcode. Tuttavia la maggioranza delle istruzioni è conforme ai formati della figura.

I primi 2 bit di ciascuna istruzione concorrono a determinare il formato dell'istruzione e indicano all'hardware dove trovare il resto dell'opcode, se presente. Nel formato 1a entrambi i registri sono sorgenti; nel formato 1b una sorgente è un registro, l'altra è una costante compresa tra – 4096 e +4095. È il bit 13 a operare la scelta tra i due formati (il bit 0 è quello più a destra). In entrambi i casi la destinazione è comunque un registro. La codifica è sufficiente per supportare 64 istruzioni, alcune delle quali sono riservate per uso futuro.

Non è possibile includere una costante di 32 bit all'interno d'istruzioni di 32 bit. L'istruzione `SETHI` imposta i primi 22 bit e lascia a un'altra istruzione il compito di impostare i rimanenti 10. È l'unica istruzione a usare questo formato.

I salti condizionati senza predizione usano il formato 3, dove il campo `COND` specifica il tipo di condizione da esaminare. Il bit `A` serve a evitare intervalli di ritardo in talune condizioni. Come già menzionato, i salti con predizione usano lo stesso formato, ma hanno spiazzamento di 19 bit.

L'ultimo formato è l'istruzione `CALL`, usata per le chiamate di procedura. Questa istruzione è speciale, perché è l'unica che utilizza 30 bit di dati per specificare un indirizzo ed è individuata da un opcode unico di 2 bit. L'indirizzo è l'indirizzo di destinazione diviso per quattro, il che consente d'indirizzare grosso modo i byte nell'intervallo ± 2^{31} relativo all'istruzione corrente.

5.3.5 Formati delle istruzioni di 8051

L'8051 dispone di sei semplici formati d'istruzioni, come illustrato nella Figura 5.16. Le istruzioni sono lunghe 1, 2 o 3 byte. Il primo formato è costituito solamente da un opcode ed è usato, per esempio, dall'istruzione che incrementa l'accumulatore.

Figura 5.16 Formati d'istruzione di 8051.

Anche il formato 2 è di un byte e consiste in un opcode di 5 bit e in un numero di registro di 3 bit. Tra le istruzioni che usano questo formato ce ne sono molte che svolgono operazioni che coinvolgono l'accumulatore e un registro, come la somma di un registro all'accumulatore o il trasferimento di dati da un registro all'accumulatore.

Il formato 3 contiene un operando di un byte che può essere per esempio una costante immediata da caricare nell'accumulatore, un offset, l'ampiezza di un salto, o anche la posizione di un bit da asserire, azzerare o esaminare.

I formati 4 e 5 sono usati nei salti e nelle chiamate a subroutine. La versione con indirizzo di 11 bit è utile quando non si dispone di memoria esterna e così tutti gli indirizzi di programma sono minori di 2096 (nell'8051) o di 8192 (nell'8052). Quando si usano più di 8 KB di memoria esterna si rende necessaria la versione a 16 bit.

Il formato 6 contiene due operandi di 8 bit ed è usato da una varietà d'istruzioni quali, per esempio, il trasferimento di una costante immediata verso un indirizzo della memoria su chip.

5.4 Indirizzamento

Molte istruzioni contengono operandi e si pone il problema di come specificarne la posizione. L'**indirizzamento** è l'argomento che tratta di queste problematiche che ora affrontiamo.

5.4.1 Modalità d'indirizzamento

Finora ci siamo curati molto poco di come vengono interpretati i bit dei campi d'indirizzo per trovare gli operandi. È giunto ora il momento di approfondire l'argomento della modalità d'indirizzamento.

5.4.2 Indirizzamento immediato

Il modo più semplice con cui un'istruzione può specificare un operando è di contenere, nel campo riservato al suo indirizzo, l'operando stesso invece che un indirizzo o qualunque altra informazione che ne descriva la posizione. Un operando così specificato si dice **immediato**, poiché viene recuperato automaticamente dalla memoria nello stesso momento in cui viene effettuato il fetch dell'istruzione; dunque è immediatamente disponibile all'uso. La Figura 5.17 mostra una possibile istruzione immediata per il caricamento della costante 4 nel registro `R1`.

L'indirizzamento immediato ha la virtù di non richiedere un riferimento supplementare in memoria per effettuare il fetch dell'operando. Presenta però lo svantaggio di poter fornire un solo operando per volta; inoltre, l'entità del valore è limitata dalla dimensione del campo. Ciononostante è una tecnica in uso presso molte architetture per la specifica di piccole costanti intere.

Figura 5.17 Istruzione per il caricamento della costante 4 nel registro R1.

5.4.3 Indirizzamento diretto

Un metodo per specificare un operando in memoria è darne l'indirizzo completo. Questa modalità si chiama **indirizzamento diretto**. Al pari di quello immediato, l'indirizzamento diretto presenta alcune limitazioni: l'istruzione accederà sempre alla stessa locazione di memoria. Se da una parte il valore contenuto può cambiare, la locazione non può. Per questa ragione l'indirizzamento diretto serve solo ad accedere a variabili globali il cui indirizzo è noto in fase di compilazione. Nonostante ciò, tale modalità è molto usata, perché molti programmi definiscono variabili globali. Nel seguito considereremo il modo in cui il computer stabilisce quali indirizzi sono immediati e quali diretti.

5.4.4 Indirizzamento a registro

L'indirizzamento a registro è concettualmente analogo all'indirizzamento diretto, ma specifica un registro invece di una locazione di memoria. Si tratta della modalità d'indirizzamento di gran lunga più utilizzata nella quasi totalità dei computer, dato che i registri sono veloci in accesso e hanno indirizzi brevi. Molti compilatori si sforzano di prevedere quali variabili saranno richiamate più spesso (per esempio gli indici di ciclo) e le destinano ai registri.

Questa modalità d'indirizzamento è nota semplicemente come **modalità a registro**. Nelle architetture load/store quali l'UltraSPARC III, quasi tutte le istruzioni usano esclusivamente questa modalità d'indirizzamento. L'unico caso in cui non viene usata è quando un operando è trasferito dalla memoria in un registro (istruzione `LOAD`) o da un registro in memoria (istruzione `STORE`). Ma anche in questi casi uno degli operandi è un registro contenente l'indirizzo della parola di memoria in lettura o scrittura.

5.4.5 Indirizzamento a registro indiretto

In questa modalità l'operando in esame proviene o è destinato alla memoria, ma il suo indirizzo non è incorporato nell'istruzione, come nel caso dell'indirizzamento diretto: l'indirizzo è contenuto in un registro. Quando un indirizzo è usato in questa maniera prende il nome di **puntatore**. Un grande vantaggio dell'indirizzamento a registro indiretto è che può referenziare la memoria senza dover necessariamente incorporare un intero indirizzo di memoria all'interno dell'istruzione. Per di più è anche possibile usare diverse parole di memoria in occasione di esecuzioni diverse della stessa istruzione.

Per capire perché può essere utile indirizzare una parola diversa a ogni esecuzione, immaginate un ciclo che passi in rassegna i 1024 elementi di un vettore d'interi e calcoli la loro somma nel registro `R1`. Altri due registri, poniamo `R2` e `R3`, sono usati fuori dal ciclo per puntare rispettivamente al primo elemento dell'array e all'indirizzo immediatamente successivo all'ultimo elemento dell'array. Se l'array comincia all'indirizzo *A* ed è composto da 1024 interi di 4 byte ciascuno, il primo indirizzo dopo l'array sarà *A* + 4096. Un codice assemblativo tipico per un calcolo del genere su una macchina a due indirizzi è riportato nella Figura 5.18.

In questo semplice programma ci avvaliamo di parecchie modalità d'indirizzamento. Le prime tre istruzioni usano la modalità a registro per il primo operando (la destinazione) e la modalità immediata per il secondo operando (una costante denotata dal simbolo #). La seconda istruzione copia l'*indirizzo* di *A* in `R2`, non il suo contenuto, il che è specificato al-


```
        MOV R1,#0          ; aggiorna la somma in R1, posto inizialmente a zero
        MOV R2,#A          ; R2 = indirizzo dell'array A
        MOV R3,#A+4096     ; R3 = indirizzo della prima parola dopo A
CICLO:  ADD R1,(R2)        ; recupera l'operando attraverso R2, registro indiretto
        ADD R2,#4          ; incrementa R2 di una parola (4 byte)
        CMP R2,R3          ; abbiamo già finito?
        BLT CICLO          ; se R2 < R3 non abbiamo finito, quindi si continua
```

Figura 5.18 Generico programma assemblativo per il calcolo della somma degli elementi di un array.

l'assembler tramite il simbolo #. Allo stesso modo la terza istruzione copia in `R2` l'indirizzo della prima parola oltre l'array.

È interessante notare che il ciclo vero e proprio non contiene nessun indirizzo di memoria. La quarta istruzione usa le modalità a registro e quella a registro indiretto; la quinta usa la modalità a registro e quella immediata; la sesta usa due volte la modalità a registro. L'istruzione `BLT` potrebbe usare un indirizzo di memoria, ma è più probabile che specifichi l'indirizzo del salto con uno spiazzamento di 8 bit relativo a se stessa. Il rifiuto totale di utilizzare indirizzi di memoria ha prodotto un ciclo conciso e veloce. Detto per inciso, questo è un vero programma per il Pentium 4, laddove abbiamo rinominato le istruzioni e i registri e modificato la notazione per renderlo di più facile lettura. Infatti la sintassi standard del linguaggio assemblativo del Pentium 4 (MASM) rasenta l'assurdo a causa del retaggio lasciatogli dall'8088, suo antenato diretto.

Facciamo notare che, in teoria, esiste un'altra soluzione per svolgere questa computazione, che non usa l'indirizzamento a registro indiretto: il ciclo avrebbe dovuto contenere un'istruzione per sommare *A* a `R1`, quale

```
ADD R1, A
```

dopodiché, a ogni iterazione del ciclo, l'istruzione stessa potrebbe venire incrementata di 4, di modo che dopo una sola iterazione diventi

```
ADD R1, A+4
```

e così via fino alla fine del ciclo.

Un programma che modifica se stesso si dice un programma **auto-modificante**. L'idea non fu di altri se non di John Von Neumann e poteva essere sensata sui primi computer, che non disponevano d'indirizzamento a registro indiretto. Oggigiorno i programmi automodificanti sono considerati di pessimo stile e difficili da capire. Inoltre non sono condivisibili da processi diversi e non funzionerebbero nemmeno su macchine con una cache di primo livello separata, laddove la I-cache (cache delle istruzioni) non disponesse di circuiti per i write-back (proprio perché i progettisti dell'hardware avrebbero supposto che i programmi non si automodifichino).

5.4.6 Indirizzamento indicizzato

Spesso è auspicabile poter referenziare una parola di memoria che si trova a un dato spiazzamento dal contenuto di un registro. Abbiamo visto qualche esempio con l'IJVM in cui le variabili locali sono referenziate specificando il loro spiazzamento rispetto a `LV`. L'indiriz-

```
        MOV R1,#0          ; aggiorna gli OR in R1, posto inizialmente a zero
        MOV R2,#0          ; R2 = indice i del prodotto corrente: A[i] AND B[i]
        MOV R3,#4096       ; R3 = indice del primo valore da non considerare
CICLO:  MOV R4,A(R2)       ; R4 = A[i]
        AND R4,B(R2)       ; R4 = A[i] AND B[i]
        OR R1,R4           ; OR di tutti i prodotti booleani in R1
        ADD R2,#4          ; i = i + 4 (passi di 1 parola = 4 byte alla volta)
        CMP R2,R3          ; abbiamo già finito?
        BLT CICLO          ; se R2 < R3 non abbiamo finito, quindi si continua
```

Figura 5.19 Generico programma assemblativo per il calcolo dell'OR di A_i AND B_i (A e B sono array di 1024 elementi).

zamento alla memoria che si ottiene specificando un registro (in via esplicita o implicita), più uno spiazzamento costante, si dice **indirizzamento indicizzato**.

Gli accessi alle variabili locali dell'IJVM usano un puntatore alla memoria (LV) contenuto in un registro, più un piccolo spiazzamento contenuto nell'istruzione stessa, come mostrato dalla Figura 4.19(a). Tuttavia è possibile anche la soluzione opposta: tenere il puntatore alla memoria nell'istruzione e il piccolo offset in un registro. Per mostrare il funzionamento di questa modalità consideriamo un esempio. Supponiamo di avere due vettori, A e B, di 1024 parole ciascuno e di voler calcolare A_i AND B_i per tutti i componenti e poi fare l'OR di questi 1024 prodotti logici per verificare se c'è almeno una coppia di componenti che non sia nulla. Sarebbe possibile salvare gli indirizzi di A e B in due registri e poi visitare gli array in stretta successione, un elemento per volta, in modo analogo a quanto esposto nella Figura 5.18. Benché sia una soluzione corretta, possiamo fare di meglio e usare uno schema più generale, come illustrato nella Figura 5.19.

Il funzionamento di questo programma è semplice. C'è bisogno di quattro registri:

1. R1 – contiene l'OR cumulativo dei prodotti logici
2. R2 – l'indice i usato per la visita degli array
3. R3 – la costante 4096, che è il primo valore di i da non considerare
4. R4 – un registro di lavoro per mantenere il calcolo di ogni prodotto.

Dopo l'inizializzazione dei registri comincia il ciclo di sei istruzioni. La prima, etichettata come CICLO, effettua il fetch di A_i in R4 con modalità indicizzata: il registro R2 viene sommato all'indirizzo A e la somma è usata come riferimento in memoria, sebbene non venga memorizzata in nessun registro visibile all'utente. La notazione

```
MOV R4,A(R2)
```

indica che la destinazione, R4, è usata in modalità registro, mentre la sorgente usa la modalità indicizzata con offset A e registro R2. Se A vale, poniamo, 124300, l'aspetto reale di questa istruzione macchina è qualcosa di simile a quanto mostrato nella Figura 5.20.

Alla prima iterazione del ciclo, R2 vale 0 (è stato così inizializzato), perciò la parola di memoria indicizzata è A_0, all'indirizzo 124300, e questa viene salvata in R4. All'iterazione successiva R2 vale 4, perciò la parola di memoria indicizzata è A_1 all'indirizzo 124304, e così via.

MOV	R4	R2	124300

Figura 5.20 Rappresentazione di MOV R4,A(R2).

Come avevamo anticipato, in questo esempio l'offset che si trova nell'istruzione è in realtà il puntatore alla memoria e il valore contenuto nel registro è un piccolo intero che viene incrementato durante il calcolo. Questa forma ovviamente richiede che nell'istruzione ci sia un campo offset abbastanza grande da contenere un indirizzo, perciò è meno efficiente dell'altra alternativa. Nondimeno si rivela spesso la scelta migliore.

5.4.7 Indirizzamento indicizzato esteso

Alcune macchine dispongono della cosiddetta modalità d'**indirizzamento indicizzato esteso**, in cui l'indirizzo di memoria è calcolato sommando tra loro il contenuto di due registri più un offset (opzionale). Un registro funge da base e l'altro da indice. Questa modalità ci sarebbe stata utile nell'esempio precedente, perché avremmo potuto inizializzare R5 con A e R6 con B fuori dal ciclo. Così facendo avremmo rimpiazzato l'istruzione all'etichetta CICLO e quella successiva con

```
CICLO: MOV R4,(R2+R5)
       AND R4,(R2+R6)
```

Poter disporre di una modalità d'indirizzamento indiretto tramite la somma di due registri e senza offset sarebbe l'ideale. Alternativamente, anche un'istruzione con un offset di 8 bit costituirebbe un miglioramento rispetto al codice originario, perché potremmo sempre porre entrambi gli offset a 0. D'altra parte, se gli offset fossero di 32 bit, non avremmo guadagnato nulla usando questa modalità. Nella pratica, le macchine che dispongono di questa modalità sono corredate della forma con offset di 8 o 16 bit.

5.4.8 Indirizzamento a stack

Abbiamo già sottolineato che è molto consigliabile rendere le istruzioni macchina quanto più corte possibile. Il limite alla riduzione della lunghezza degli indirizzi equivale a non averne per nulla. Come già visto nel Capitolo 4 le istruzioni senza indirizzi, come l'istruzione IADD, sono possibili in associazione con uno stack. In questo paragrafo analizziamo più da vicino l'indirizzamento a stack.

Notazione polacca inversa

È tradizione in matematica indicare l'operatore in mezzo agli operandi, come in $x + y$, invece che dopo gli operandi, come in $x\,y+$. La forma con l'operatore "in" mezzo è detta notazione **infissa**, mentre la forma con l'operatore dopo gli operandi si chiama **postfissa** o anche **notazione polacca inversa**, dal logico polacco J. Lukasiewicz (1958) che ne studiò le proprietà.

La notazione polacca inversa presenta un certo numero di vantaggi rispetto all'infissa in merito alla scrittura di formule algebriche. Per prima cosa qualsiasi formula può essere espressa correttamente senza parentesi. Inoltre la valutazione delle formule in tale notazione si addice

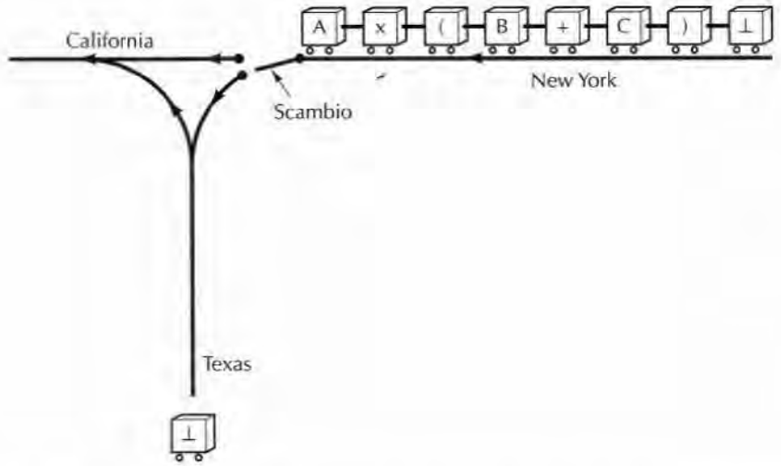


Figura 5.21 Ogni vagone ferroviario rappresenta un simbolo della formula infissa da convertire in notazione polacca inversa.

particolarmente ai compilatori con stack. Infine, per gli operatori infissi si definisce una precedenza, che è arbitraria e poco gradita. Per esempio sappiamo che $a \times b + c$ vuol dire $(a \times b) + c$ e non $a \times (b + c)$ perché alla moltiplicazione è stata assegnata, in modo arbitrario, una precedenza maggiore rispetto alla somma. Ma lo scorrimento verso sinistra ha precedenza sull'AND booleano? Chi può dirlo? La notazione polacca inversa elimina questa seccatura.

Esistono molteplici algoritmi per convertire formule infisse in notazione polacca inversa. Quella fornita qui è un riadattamento di un'idea di E. W. Dijkstra. Supponete che una formula sia composta dai simboli seguenti: le variabili, gli operatori binari (a due operandi) + – × / e le parentesi destra e sinistra. Il simbolo ⊥ delimita la formula: è anteposto al suo primo simbolo e segue l'ultimo.

La Figura 5.21 mostra un tracciato ferroviario che collega New York alla California, con al centro una diramazione che si diparte in direzione del Texas. A ogni simbolo della formula corrisponde un vagone ferroviario. Il treno procede verso ovest (verso sinistra) e ogni volta che un vagone raggiunge lo scambio deve fermarsi e stabilire se procedere direttamente verso la California o passare prima per il Texas. I vagoni contenenti le variabili procedono sempre diretti per la California, mentre quelli contenenti altri simboli, prima di entrare nello scambio, devono ispezionare il contenuto del vagone più vicino che si trova lungo la diramazione per il Texas.

La tabella nella Figura 5.22 elenca le situazioni che si possono verificare, a seconda del vagone fermo allo scambio e del vagone a lui più vicino lungo il tracciato che porta in Texas. Il primo ⊥ va sempre in Texas. I numeri della figura rimandano ai seguenti eventi:

1. il vagone allo scambio procede verso il Texas
2. il vagone che per ultimo ha intrapreso la linea per il Texas inverte il percorso e prosegue per la California

Vagone fermo allo scambio

Vagone entrato più di recente nella linea per il Texas	⊥	+	–	×	/	(	)
⊥	4	1	1	1	1	1	5
+	2	2	2	1	1	1	2
–	2	2	2	1	1	1	2
×	2	2	2	2	2	1	2
/	2	2	2	2	2	1	2
(	5	1	1	1	1	1	3

Figura 5.22 Tavola di decisione utilizzata dall'algoritmo per la conversione da notazione infissa a polacca inversa.

3. il vagone allo scambio e quello che è entrato più di recente nella diramazione per il Texas vengono entrambi dirottati altrove e scompaiono (cioè si elidono)
4. stop: i simboli presenti al momento in California rappresentano la notazione polacca inversa della formula se letti da sinistra verso destra
5. stop: è stato rilevato un errore sintattico nella formula originaria.

Al termine di ogni azione intrapresa viene effettuato un nuovo confronto tra il vagone attualmente fermo allo scambio, che potrebbe essere lo stesso del confronto precedente o il successivo, e il vagone che per ultimo è proseguito per il Texas. Il processo continua finché non si raggiunge l'evento 4. Notate che la linea per il Texas è usata come uno stack, dove l'instradamento di un vagone verso il Texas corrisponde a un'operazione di push, mentre l'inversione di marcia di un vagone che si trova già in Texas e che prosegue per la California corrisponde a un'operazione di pop.

L'ordine delle variabili è lo stesso nelle due notazioni, invece l'ordine degli operatori può cambiare. Nella notazione polacca inversa gli operatori compaiono nell'ordine in cui verranno effettivamente eseguiti durante la valutazione dell'espressione. La Figura 5.23 fornisce diversi esempi di formule infisse e delle stesse in notazione polacca inversa.

Notazione infissa	Notazione polacca inversa
A + B × C	A B C × +
A × B + C	A B × C +
A × B + C × D	A B × C D × +
(A + B) / (C – D)	A B + C D – /
A × B / C	A B × C /
((A + B)× C + D) / (E + F + G)	A B + C × D + E F + G + /

Figura 5.23 Esempi di espressioni in notazione infissa e le stesse in notazione polacca inversa.

Valutazione delle formule in notazione polacca inversa

La notazione polacca inversa è la notazione ideale per la valutazione di una formula da parte di un computer dotato di stack. Una formula è costituita da *n* simboli, ciascuno dei quali è un operando o un operatore. L'algoritmo che si avvale di uno stack per la valutazione di una formula in notazione polacca inversa è molto semplice: scorre la stringa da sinistra verso destra e, quando incontra un operando, lo impila sullo stack. Invece quando incontra un operatore ne esegue l'istruzione corrispondente.

La Figura 5.24 illustra la valutazione della formula

```
( 8 + 2 × 5 ) / ( 1 + 3 × 2 − 4 )
```

nell'IJVM. La stessa formula espressa in notazione polacca inversa è

```
8 2 5 × + 1 3 2 × + 4 − /
```

Nella figura abbiamo introdotto le istruzioni `IMUL` e `IDIV`, rispettivamente di moltiplicazione e di divisione. Il numero sulla cima dello stack è l'operando destro, non quello sinistro; questa precisazione è importante perché nella divisione e nella sottrazione l'ordine degli operandi conta (a differenza dell'addizione e della moltiplicazione). In altre parole, `IDIV` è stata definita appositamente in modo tale che, dopo aver fatto il push del numeratore e poi quello del denominatore, la sua esecuzione produca come risultato la divisione corretta. Si noti la semplicità di generazione del codice per l'IJVM: si scorre la notazione polacca inversa della formula e si restituisce un'istruzione per ciascun simbolo. Se il simbolo è una costante o una variabile, si restituisce un'istruzione di push sullo stack; se il simbolo è un operatore, si restituisce l'istruzione che esegue l'operazione.

Passo	Stringa rimanente	Istruzione	Stack
1	8 2 5 × + 1 3 2 × + 4 − /	BIPUSH 8	8
2	2 5 × + 1 3 2 × + 4 − /	BIPUSH 2	8, 2
3	5 × + 1 3 2 × + 4 − /	BIPUSH 5	8, 2, 5
4	× + 1 3 2 × + 4 − /	IMUL	8, 10
5	+ 1 3 2 × + 4 − /	IADD	18
6	1 3 2 × + 4 − /	BIPUSH 1	18, 1
7	3 2 × + 4 − /	BIPUSH 3	18, 1, 3
8	2 × + 4 − /	BIPUSH 2	18, 1, 3, 2
9	× + 4 − /	IMUL	18, 1, 6
10	+ 4 − /	IADD	18, 7
11	4 − /	BIPUSH 4	18, 7, 4
12	− /	ISUB	18, 3
13	/	IDIV	6

Figura 5.24 Uso dello stack per la valutazione di una formula in notazione polacca inversa.

5.4.9 Modalità d'indirizzamento per istruzioni di salto

Fin qui abbiamo analizzato le istruzioni che operano sui dati. Anche le istruzioni di salto (e le chiamate di procedura) necessitano di modalità d'indirizzamento per specificare l'indirizzo di destinazione. Le modalità riscontrate finora funzionano grosso modo anche per i salti. L'indirizzamento diretto è di certo un'opzione possibile, secondo cui l'indirizzo di destinazione viene semplicemente riportato per intero all'interno dell'istruzione.

Esistono anche altre modalità d'indirizzamento sensate. L'indirizzamento a registro indiretto consente al programma di calcolare l'indirizzo di destinazione, scriverlo in un registro e quindi effettuare il salto. È la modalità più flessibile perché l'indirizzo di destinazione può essere calcolato durante l'esecuzione, ma è anche la più incline alla generazione di bachi quasi impossibili da scovare.

Un'altra modalità ragionevole è la modalità indicizzata, che specifica un certo offset rispetto all'indirizzo contenuto in un registro. Manifesta le stesse proprietà della modalità a registro indiretto.

Un'ulteriore opzione è l'indirizzamento relativo al `PC`: un offset (con segno) contenuto nell'istruzione stessa viene sommato al program counter per ottenere l'indirizzo di destinazione. In realtà non è altro che una modalità indicizzata che usa il `PC` come registro base.

5.4.10 Ortogonalità dei codici operativi e delle modalità d'indirizzamento

Dal punto di vista software, le istruzioni e l'indirizzamento dovrebbero manifestare una struttura regolare, con un numero minimo di formati d'istruzioni. Tale struttura agevolerebbe molto il compilatore a produrre codice di qualità. Gli opcode dovrebbero consentire tutte le modalità d'indirizzamento, laddove sensato, e ogni registro (anche `FP`, `SP` e `PC`) dovrebbe essere utilizzabile da tutte le modalità a registro.

Un esempio di progetto elegante per una macchina a tre indirizzi prevede i formati d'istruzioni di 32 bit della Figura 5.25 e supporta fino a 256 codici operativi. Il formato 1 prevede due indirizzi sorgente e un indirizzo destinazione ed è usato da tutte le istruzioni logico-aritmetiche.

Il campo finale di 8 bit è inutilizzato e può essere destinato per differenziare ulteriormente le istruzioni. A mo' d'esempio, un solo opcode potrebbe specificare l'insieme di ope-

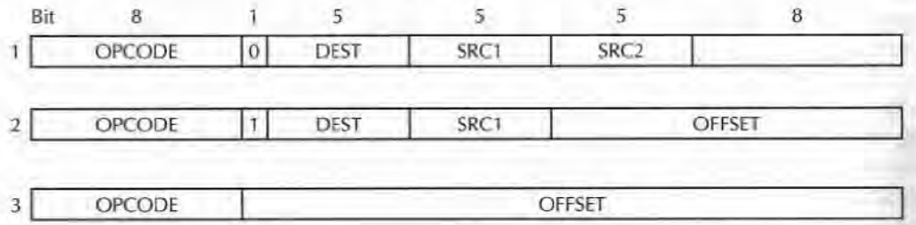


Figura 5.25 Semplice progetto dei formati d'istruzione di una macchina a tre indirizzi.

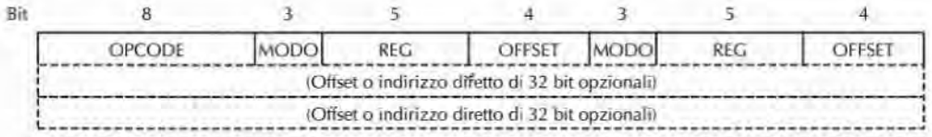


Figura 5.26 Semplice progetto dei formati d'istruzione di una macchina a due indirizzi.

razioni in virgola mobile, e il campo supplementare potrebbe distinguerle l'una dall'altra. Proseguendo, se il bit 23 è asserito, l'istruzione usa il formato 2, in cui il secondo operando non è più un registro, bensì una costante immediata con segno di 13 bit. È questo un formato confacente alle istruzioni `LOAD` e `STORE` che referenziano la memoria in modo indicizzato.

Occorrono poche altre istruzioni, come i salti condizionati, facilmente adattabili al formato 3. Per esempio tutte le istruzioni di salto (condizionato), di chiamata di procedura, e così via, potrebbero avere un proprio opcode, il che lascia 24 bit a disposizione per l'offset relativo al `PC`. Se l'offset è interpretato come numero di parole, allora abbraccia un intervallo di ± 32 MB. Si potrebbe anche riservare un certo numero di opcode per istruzioni `LOAD` e `STORE` che necessitano dell'offset lungo del formato 3. Si tratterebbe d'istruzioni non del tutto generali (che per esempio potrebbero agire per definizione solo sul registro `R0`), ma il loro uso sarebbe alquanto limitato.

Si consideri ora il progetto di una macchina a due indirizzi, mostrata nella Figura 5.26, che può specificare parole di memoria per entrambi gli operandi. Una tale macchina è in grado di sommare una parola di memoria a un registro, oppure sommare tra loro due parole di memoria. Allo stato attuale gli accessi in memoria sono relativamente gravosi, perciò questo progetto non è quasi mai preso in considerazione. Se i progressi tecnologici renderanno in futuro gli accessi in memoria e alla cache più convenienti, risulterà un progetto particolarmente semplice ed efficiente. Le macchine PDP-11 e VAX ottennero un grande successo e dominarono la scena informatica per vent'anni proprio grazie a un progetto di questo tipo.

Anche in questo schema gli opcode sono di 8 bit, ma questa volta disponiamo di 12 bit per specificare la sorgente e di altri 12 per la destinazione. Ciascun operando è corredato di 3 bit per la modalità, 5 bit per il registro e 4 per l'offset. Con 3 bit potremmo gestire tutte le modalità, immediata, diretta, a registro, a registro indiretto, indicizzata, a stack, e avremmo ancora spazio per altre due modalità. È questo un progetto semplice ed elegante, assicura una facile compilazione ed è abbastanza flessibile, soprattutto se il program counter, il puntatore allo stack e il puntatore alle variabili locali sono tutti accessibili come registri d'uso generale.

L'unico problema che si presenta è che per l'indirizzamento diretto abbiamo bisogno di un numero di bit maggiore per gli indirizzi. La soluzione del PDP-11 e del VAX era di aggiungere all'istruzione una parola supplementare per ogni indirizzo di operando indirizzato direttamente. Potremmo anche usare una delle due modalità d'indirizzamento libere per una modalità indicizzata con un offset di 32 bit posposto all'istruzione. Così facendo, una somma di due operandi in memoria, entrambi indirizzati direttamente o con la lunga forma indicizzata, prenderebbe nel caso peggiore 96 bit e userebbe tre cicli di bus (uno per l'istruzione, due per gli operandi). D'altra parte quasi tutte le architetture RISC richiederebbero almeno 96 bit, se non di più, per sommare due parole di memoria qualsiasi, e userebbero almeno quattro cicli di bus.

Le alternative alla Figura 5.26 sono molte. Questo progetto rende possibile l'esecuzione dell'istruzione

```
i = j;
```

con una sola istruzione di 32 bit, ammesso che *i* e *j* si trovino entrambe tra le prime 16 variabili locali. Per variabili oltre la sedicesima bisogna optare però per gli offset di 32 bit. Un'alternativa sarebbe quella di avere un altro formato con un solo offset di 8 bit invece di due offset di 4 bit, più una regola per stabilire se si riferisce alla sorgente o alla destinazione, ma non a entrambe. Le possibilità sono illimitate così come i compromessi, e i progettisti di una macchina devono giocare su molti fattori per ottenere un buon risultato.

5.4.11 Modalità d'indirizzamento del Pentium 4

Le modalità d'indirizzamento del Pentium 4 sono assai irregolari e cambiano a seconda che una certa istruzione sia in modalità di 16 o di 32 bit. Non affronteremo la modalità di 16 bit, dato che quella di 32 è già abbastanza intricata. Il Pentium prevede le modalità immediata, diretta, a registro, a registro indiretto, più una speciale per l'indirizzamento di elementi di un array. Il problema è che non tutte le modalità si applicano a tutte le istruzioni, e non tutti i registri sono utilizzabili da tutte le modalità, il che accresce la difficoltà delle mansioni del compilatore e porta a un codice più scadente.

Il byte `MODE` della Figura 5.14 controlla le modalità d'indirizzamento. Uno degli operandi è specificato in combinazione dai campi `MOD` e `R/M`, mentre l'altro è sempre un registro, specificato dal valore del campo `REG`. La Figura 5.27 elenca le 32 combinazioni che possono essere specificate congiuntamente dai 2 bit di `MOD` e dai 3 bit di `R/M`. Per esempio quando entrambi i campi valgono zero l'operando viene letto all'indirizzo di memoria contenuto nel registro `EAX`.

Le colonne 01 e 10 hanno a che fare con modalità in cui il registro è sommato a un offset di 8 o 32 bit posposto all'istruzione. Se l'offset è di 8 bit, viene portato a 32 (estensione del segno) prima della somma. Per fare un esempio, un'istruzione `ADD`, con `R/M` = 011,

	MOD			
R/M	**00**	**01**	**10**	**11**
000	M[EAX]	M[EAX + OFFSET8]	M[EAX + OFFSET32]	EAX o AL
001	M[ECX]	M[ECX + OFFSET8]	M[ECX + OFFSET32]	ECX o CL
010	M[EDX]	M[EDX + OFFSET8]	M[EDX + OFFSET32]	EDX o DL
011	M[EBX]	M[EBX + OFFSET8]	M[EBX + OFFSET32]	EBX o BL
100	SIB	SIB con OFFSET8	SIB con OFFSET32	ESP o AH
101	Diretto	M[EBP + OFFSET8]	M[EBP + OFFSET32]	EBP o CH
110	M[ESI]	M[ESI + OFFSET8]	M[ESI + OFFSET32]	ESI o DH
111	M[EDI]	M[EDI + OFFSET8]	M[EDI + OFFSET32]	EDI o BH

Figura 5.27 Modalità d'indirizzamento a 32 bit del Pentium 4. M[x] è la parola di memoria all'indirizzo *x*.

MOD = 01 e offset pari a 6, calcola la somma di EBX più 6 e usa il risultato come indirizzo della parola di memoria contenente un operando. EBX non ne risulta modificato.

La colonna MOD = 11 consente di specificare due registri alla volta: il primo è usato per le istruzioni di una parola, il secondo per le istruzioni di un byte. Osservate la parziale irregolarità della tabella: per esempio non c'è modo di usare EBP in modalità indiretta, né di usare ESP come base di un offset.

In alcune modalità c'è un byte aggiuntivo, detto SIB, che segue il byte MODE (Figura 5.14). Il byte SIB specifica due registri e un fattore di scala. Quando il byte SIB è presente, l'indirizzo dell'operando si calcola moltiplicando l'indirizzo indice per 1, 2, 4 o 8 (a seconda della scala), sommando il risultato al registro base; infine, c'è la possibilità che il tutto vada sommato a uno spiazzamento di 8 o 32 bit, secondo quanto specidicato da MOD. Pressoché tutti i registri possono essere usati come indice o come base.

Le modalità SIB sono utili per accedere a elementi di un array; considerate per esempio l'istruzione Java

```
for (i = 0; i < n; i++) a[i] = 0;
```

dove a è un array d'interi di 4 byte locale alla procedura corrente. In genere EBP è usato per puntare alla base del record d'attivazione dello stack contenente le variabili e gli array locali, come mostrato nella Figura 5.28. Il compilatore potrebbe mantenere *i* in EAX e, per accedere ad a[i], userebbe una modalità SIB in cui l'indirizzo dell'operando si otterrebbe sommando 4 × EAX a EBP e a 8. L'istruzione Java potrebbe così assegnare un valore ad a[i] con una sola istruzione di STORE.

Il gioco vale la candela? Difficile a dirsi. Sicuramente questa istruzione, se usata propriamente, fa risparmiare qualche ciclo. La frequenza del suo utilizzo dipende dal compilatore e dall'applicazione. Il problema è che questa istruzione occupa dello spazio sul chip che si sarebbe potuto impiegare in modo diverso se non fosse esistita. Per esempio la cache di primo livello avrebbe potuto essere più grande, o il chip più piccolo, consentendo forse una velocità di clock leggermente superiore.

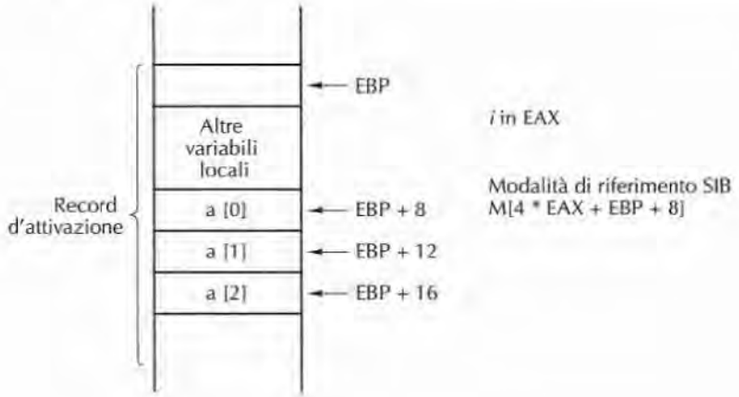


Figura 5.28 Accesso ad a[i].

Sono questi i compromessi sui quali i progettisti sono spesso chiamati a pronunciarsi. Di norma vengono eseguite numerose simulazioni prima dell'implementazione sul silicio, ma le simulazioni sono indicative solo se si ha già una buona idea di quel che sarà il carico di lavoro. Si può scommettere sul sicuro che i progettisti dell'8088 non abbiano incluso i browser di Internet nel loro insieme di prova. Ciononostante, alcuni discendenti di quel prodotto sono oggi usati principalmente per la navigazione del Web e così le scelte fatte vent'anni fa potrebbero rivelarsi completamente inadeguate per le applicazioni correnti. Ancora una volta in nome della retrocompatibilità, quando una caratteristica entra in un progetto è impossibile estrometterla.

5.4.12 Modalità d'indirizzamento di UltraSPARC III

Nell'ISA di UltraSPARC tutte le istruzioni si avvalgono dell'indirizzamento immediato o a registro, eccezion fatta per quelle che indirizzano la memoria. Nella modalità a registro sono i 5 bit a specificare quale indirizzo usare. Nella modalità immediata, i dati sono contenuti in una costante (con segno) di 13 bit. Non ci sono altre modalità disponibili per le istruzioni logico-aritmetiche o simili.

Ci sono tre tipi d'istruzioni che indirizzano la memoria: LOAD, STORE e un'istruzione di sincronizzazione multiprocessore. Le istruzioni LOAD e STORE possono indirizzare la memoria in due modalità diverse: nella prima si usa la somma di due registri come valore d'indirizzamento indiretto; la seconda è l'indirizzamento tradizionale, che si avvale di un offset (con segno) di 13 bit.

5.4.13 Modalità d'indirizzamento di 8051

L'8051 presenta una struttura d'indirizzamento abbastanza regolare. Ci sono poche modalità basilari, la più semplice delle quali è una modalità implicita basata sull'accumulatore. Molte istruzioni combinano un operando con l'accumulatore, per esempio li sommano, li sottraggono, ne fanno l'AND o l'OR. Queste istruzioni non hanno bisogno di bit supplementari per indicare il fatto che usano l'accumulatore: è l'opcode a stabilirlo implicitamente.

La seconda modalità è a registro, dunque l'operando si trova in un registro. I registri possono fungere da sorgente o da destinazione. La terza modalità è l'indirizzamento diretto, secondo cui l'operando si trova in memoria presso un indirizzo contenuto nell'istruzione stessa. Nella quarta modalità, a registro indiretta, un registro contiene un puntatore all'operando; dal momento che i registri sono lunghi 8 bit, gli operand'indirizzati in questa maniera devono trovarsi nei primi 256 byte di memoria. La quinta modalità è quella immediata, secondo cui l'operando è esso stesso parte dell'istruzione.

L'8051 dispone anche di una modalità d'indirizzamento speciale, usata da un numero ristretto d'istruzioni che accedono alla memoria. Si ricordi che un 8051 può avere fino a 64 KB di memoria esterna per le istruzioni e altri 64 KB di memoria esterna per i dati: bisogna potervi accedere in qualche modo. La memoria istruzioni esterna è manipolata da due istruzioni con offset di 16 bit: LJMP, che può accedere a qualsiasi indirizzo di memoria, e LCALL, che può invocare una subroutine, in qualunque parte della memoria si trovi.

L'accesso alla memoria dati esterna è gestito diversamente. L'8051 ha un registro puntatore di 16 bit, DPTR, che può contenere un indirizzo di memoria da 16 bit. Così i pro-

grammi possono caricarlo e usarlo per referenziare indirettamente qualsiasi byte nello spazio di memoria di 64 KB.

5.4.14 Analisi delle modalità d'indirizzamento

Abbiamo dunque introdotto un certo numero di modalità d'indirizzamento. Quelle utilizzate dal Pentium 4, dall'UltraSPARC III e dall'8051 sono riassunte nella Figura 5.29. Si tenga conto però, come già sottolineato, che non tutte le istruzioni possono avvalersi della gamma completa delle modalità d'indirizzamento.

Nella pratica un ISA, per essere efficace, non ha bisogno di molte modalità d'indirizzamento. Oggigiorno la maggior parte del codice scritto a questo livello viene generato dai compilatori (a parte forse nel caso dell'8051), perciò gli aspetti più importanti delle modalità d'indirizzamento di un'architettura sono la possibilità di scegliere tra poche alternative chiare, il cui costo (in termini di tempo di calcolo e dimensioni del codice) sia valutabile prontamente. Ciò si traduce in una presa di posizione drastica: una macchina dovrebbe offrire ogni scelta possibile, oppure una sola. Ogni via di mezzo costringe il compilatore a fare delle scelte per cui non è abbastanza preparato o sofisticato.

Le architetture più eleganti dispongono in genere di un numero estremamente limitato di modalità d'indirizzamento e pongono vincoli molto stringenti al loro uso. Nella pratica, alla maggior parte delle applicazioni basta poter disporre delle modalità immediata, diretta, a registro e indicizzata. Inoltre, ogni registro dovrebbe essere utilizzabile al pari di ogni altro, compresi il puntatore al record d'attivazione, il puntatore allo stack e il program counter. Modalità d'indirizzamento più complicate possono sì ridurre il numero d'istruzioni, ma al costo dell'introduzione di sequenze di operazioni non facilmente eseguibili in modo parallelo ad altre operazioni sequenziali.

Abbiamo così completato lo studio dei compromessi possibili tra opcode e indirizzi da una parte, e forme d'indirizzamento dall'altra. Ogniqualvolta vi avvicinate a un nuovo computer dovreste esaminarne le istruzioni e le modalità d'indirizzamento, non solo per scoprire quali sono disponibili, ma anche per capire le scelte fatte e le conseguenze che avrebbero avuto le scelte alternative.

Modalità d'indirizzamento	Pentium 4	UltraSPARC III	8051
Accumulatore			×
Immediata	×	×	×
Diretta	×		×
A registro	×	×	×
A registro indiretta	×	×	×
Indicizzata	×	×	
Indicizzata estesa		×	
A stack			

Figura 5.29 Modalità d'indirizzamento a confronto.

5.5 Tipi d'istruzioni

Anche se possono differire nei dettagli, le istruzioni del livello ISA possono essere suddivise approssimativamente in una mezza dozzina di gruppi facilmente rintracciabili in tutte le macchine. Oltre a questi, ciascun computer dispone di una manciata d'istruzioni insolite, aggiunte per motivi di compatibilità con modelli precedenti o a seguito di un'idea brillante di un progettista, o perché un'agenzia governativa ha pagato il produttore perché la includesse nel progetto. Proveremo a dar conto brevemente di tutte le categorie più comuni, senza pretendere di essere esaustivi.

5.5.1 Istruzioni di trasferimento dati

Poter copiare dati da una locazione all'altra è fondamentale per tutte le operazioni. Per copia intendiamo la creazione di un nuovo oggetto costituito da una sequenza di bit identica all'originale. L'uso tecnico della parola "trasferimento" è diverso da quello del linguaggio comune. Quando diciamo che Mario Cervi si è trasferito da New York alla California, non intendiamo dire che è stata creata una copia identica di Mario Cervi in California e che l'originale si trova ancora a New York. Quando diciamo che il contenuto della locazione di memoria 2000 è stato trasferito in un qualche registro, intendiamo sempre che vi è stata creata una copia fedele e che l'originale si trova ancora indisturbato nella sua locazione 2000. Le istruzioni di trasferimento dati dovrebbero chiamarsi piuttosto istruzioni di "duplicazione dei dati", ma oramai la locuzione "trasferimento di dati" è entrata nell'uso.

Ci sono due ragioni per copiare i dati da una locazione a un'altra, una delle quali è fondamentale: l'assegnamento di valori a variabili. L'assegnamento

```
A = B
```

si implementa con la copia del valore all'indirizzo di memoria *B* nella locazione di memoria *A*, come specificato dal programmatore. La seconda motivazione per la copia dei dati è prepararli a un accesso e a un uso efficienti. Si è già visto come molte istruzioni abbiano accesso solo alle variabili contenute nei registri. Poiché i dati possono provenire da due sorgenti (la memoria o i registri) ed essere destinati a due locazioni (in memoria o nei registri), ci sono quattro tipi diversi di trasferimenti possibili. Alcuni computer dispongono di quattro istruzioni diverse, una per ogni situazione, altri hanno una sola istruzione per tutte le situazioni. Altri ancora usano `LOAD` per trasferire dalla memoria verso i registri, `STORE` dai registri alla memoria, `MOVE` per i trasferimenti tra registri e non dispongono di alcuna istruzione per la copia tra locazioni di memoria.

Le istruzioni per il trasferimento dati devono specificare in qualche maniera la quantità di dati da trasferire. In alcuni ISA esistono istruzioni per trasferire quantità di dati variabili da un byte fino all'intera memoria. Sulle macchine con parole di lunghezza fissa l'unità di trasferimento consueta è proprio la parola. Il trasferimento di quantità maggiori o minori di una parola va svolto via software per mezzo di operazioni di scorrimento e fusione. Alcuni ISA prevedono funzionalità aggiuntive per la copia sia di quantità più piccole di una parola (solitamente per multipli di byte), sia di più parole per volta. La copia di più parole è tanto più delicata quanto più grande è il massimo numero di parole trasferibili, perché un'operazione di questo genere può impiegare molto tempo e c'è il rischio che venga inter-

rotta nel bel mezzo dell'esecuzione. Alcune macchine con lunghezza di parola variabile dispongono d'istruzioni che specificano solo gli indirizzi sorgente e destinazione, senza indicare la quantità da trasferire, così la copia dei dati continua fino al raggiungimento di un marcatore di fine dati.

5.5.2 Operazioni binarie

Le operazioni binarie sono quelle che producono un risultato dalla combinazione di due operandi. Tutti gli ISA hanno istruzioni per l'addizione e la sottrazione d'interi. Anche la moltiplicazione e la divisione d'interi è pressoché standard. È abbastanza ovvio che un computer sia dotato d'istruzioni aritmetiche e non ci dilunghiamo in merito.

Un altro insieme di operazioni binarie comprende le istruzioni booleane. Benché esistano 16 funzioni booleane in due variabili, ben poche macchine (forse nessuna) dispongono d'istruzioni per tutte e 16. In genere sono disponibili AND, OR e NOT, qualche volta anche XOR (OR ESCLUSIVO), NOR e NAND.

Un uso importante di AND è l'estrazione di bit da una parola. Considerate per esempio una macchina con parole di 32 bit che possono ospitare quattro caratteri di 8 bit. Se si vuole separare il secondo carattere dagli altri tre al fine di visualizzarlo sullo schermo, è necessario creare una parola che lo contenga negli 8 bit più a destra e che contenga tutti zero nei 24 bit rimanenti, perciò detta parola **giustificata a destra**.

L'estrazione del carattere avviene facendo l'AND della parola con una costante, detta **maschera**. Il risultato di questa operazione è che tutti i bit indesiderati vengono posti a zero, vale a dire mascherati, come mostrato nello schema seguente

```
10110111 10111100 11011011 10001011  A
00000000 11111111 00000000 00000000  B (maschera)
00000000 10111100 00000000 00000000  A AND B
```

Quindi il risultato viene fatto scorrere di 16 bit verso destra in modo da isolare il carattere da estrarre all'estremità destra della parola.

Un uso importante di OR è quello di impacchettare bit in una parola, che è l'operazione inversa dell'estrazione. Per cambiare gli 8 bit meno significativi di una parola da 32 bit senza modificare gli altri, per prima cosa mascheriamo gli 8 bit indesiderati, dopodiché il nuovo carattere è inserito fancendone l'OR, come mostrato di seguito.

```
10110111 10111100 11011011 10001011 A
11111111 11111111 11111111 00000000 B (maschera)
10110111 10111100 11011011 00000000 A AND B
00000000 00000000 00000000 01010111 C
10110111 10111100 11011011 01010111 (A AND B) OR C
```

L'operazione AND tende a rimuovere i bit 1, perché il suo risultato non contiene mai più bit 1 di quanti ce ne siano in ciascun operando. L'operazione OR tende a inserire bit 1 perché il risultato contiene sempre almeno tanti bit 1 quanti ce ne sono nell'operando con numero maggiore. L'operazione XOR, d'altra parte, ha un comportamento simmetrico, perché tende a inserire in media tanti bit 1 quanti ne rimuove. Si tratta di una proprietà utile in alcune situazioni, come nella generazione di numeri casuali.

Molti dei computer odierni supportano anche un insieme d'istruzioni in virgola mobile, più o meno corrispondenti allle operazioni aritmetiche sugli interi. Gran parte delle macchine mette a disposizione almeno due formati di numeri in virgola mobile, il più corto per esigenza di velocità, l'altro per quelle situazioni particolari in cui si richiede una precisione maggiore. Ci sono molte varianti possibili per i formati in virgola mobile, eppure c'è uno standard che è oggi quasi universalmente accettato: IEEE 754. I numeri in virgola mobile, e il formato IEEE 754, sono presentati nell'Appendice B.

5.5.3 Operazioni unarie

Le operazioni unarie prendono in ingresso un operando e restituiscono un risultato. Queste istruzioni sono in genere più corte di quelle binarie, visto che contengono un indirizzo in meno, sebbene spesso richiedano la specifica di altre informazioni.

Le istruzioni per lo scorrimento (*shift*) o la rotazione del contenuto di una parola si rivelano molto utili, perciò sono spesso presenti in più varianti. Le operazioni di scorrimento possono spostare i bit verso destra o verso sinistra, e i bit che fuoriescono dalla parola si intendono perduti. Le rotazioni sono scorrimenti in cui i bit che escono da un'estremità della parola riappaiono all'altra estremità. Illustriamo con un esempio la differenza tra le due operazioni.

```
00000000 00000000 00000000 01110011  A
00000000 00000000 00000000 00011100  A scorso verso destra di 2 bit
11000000 00000000 00000000 00011100  A ruotato verso destra di 2 bit
```

Scorrimenti e rotazioni sono utili in entrambe le direzioni. Se una parola di n bit è ruotata verso sinistra di k bit, si ottiene lo stesso risultato che ruotandola verso destra di $n - k$ bit.

Gli scorrimenti verso destra sono spesso usati in associazione con l'estensione del segno, ovvero le posizioni all'estremità di sinistra lasciate vacanti dallo scorrimento vengono riempite con il bit di segno originario, 0 o 1. È come se il bit di segno venisse trascinato verso destra dallo scorrimento. Tra le altre cose, ciò implica che un numero negativo resta negativo. È quanto succede nel seguente esempio di scorrimento di 2 bit verso destra.

```
11111111 11111111 11111111 11110000 A
00111111 11111111 11111111 11111100 A scorso senza estensione del segno
11111111 11111111 11111111 11111100 A scorso con estensione del segno
```

Un'applicazione importante dello scorrimento è la moltiplicazione e la divisione per potenze di 2. Se un intero positivo viene fatto scorrere di k bit verso sinistra allora, a meno di overflow, il risultato rappresenta il numero iniziale moltiplicato per 2^k. Se un numero positivo viene fatto scorrere verso destra di k bit, si ottiene il numero iniziale diviso per 2^k.

Lo scorrimento può essere usato per accelerare certe operazioni aritmetiche. Si consideri, per esempio, il calcolo di $18 \times n$ con n intero positivo. Poiché $18 \times n = 16 \times n + 2 \times n$, è possibile ottenere $16 \times n$ facendo scorrere n verso sinistra di 4 bit, $2 \times n$ facendo scorrere n verso sinistra di 1 bit. La somma di questi due numeri è proprio $18 \times n$, così la moltiplicazione è stata realizzata tramite un trasferimento, due scorrimenti e un'addizione, il che è spesso più veloce di una moltiplicazione vera e propria. Ovviamente è uno stratagemma che il compilatore può usare solo se almeno uno dei fattori è una costante.

Tuttavia lo scorrimento di numeri negativi, anche se fatto con estensione del segno, dà risultati molto diversi. Si consideri per esempio il numero –1, in complemento a uno, che, se fatto scorrere di una posizione verso sinistra, produce –3. Un altro scorrimento di un bit verso sinistra porta a –7:

```
11111111 11111111 11111111 11111110  –1 in complemento a uno
11111111 11111111 11111111 11111100  –1 scorso verso sinistra di 1 bit = –3
11111111 11111111 11111111 11111000  –1 scorso verso sinistra di 2 bit = –7
```

Lo scorrimento verso sinistra dei numeri negativi in complemento a uno non equivale a moltiplicare per 2. Invece il loro scorrimento a destra simula la divisione correttamente.

Prendete ora la rappresentazione di –1 in complemento a due. Facendola scorrere di 6 bit verso destra si ottiene ancora –1, il che è sbagliato visto che la parte intera di –1/64 è 0:

```
11111111 11111111 11111111 11111111  –1 in complemento a due
11111111 11111111 11111111 11111111  –1 scorso di 6 bit verso destra = –1
```

In generale lo scorrimento verso destra introduce errori perché tronca il risultato. Invece lo scorrimento a sinistra simula davvero la moltiplicazione per 2.

Le operazioni di rotazione sono utili per l'impacchettamento di sequenze di bit in parole e per il loro spacchettamento. Se si vuole esaminare una parola bit per bit, la si può ruotare di 1 bit alla volta (non importa in che direzione) ed esaminare a ogni passo il contenuto del bit di segno; dopo aver esaminato tutti i bit la parola risulta ripristinata nella sua forma originale. Le operazioni di rotazione sono più genuine delle operazioni di scorrimento, perché non comportano perdita d'informazione: gli effetti di un'operazione di rotazione arbitraria possono essere annullati da un'altra operazione di rotazione.

Alcune operazioni binarie coinvolgono certi operandi così di frequente che alle volte gli ISA dispongono d'istruzioni unarie per svolgerle più velocemente. La copia del valore zero in una certa parola di memoria o in un registro è estremamente frequente nella fase di inizializzazione di una computazione. La copia di zero è naturalmente un caso speciale d'istruzione di trasferimento dati. Per ragioni di efficienza, si usa spesso l'operazione `CLR` che contiene un solo indirizzo, quello della locazione da azzerare (*clear*).

La somma di una parola con il valore 1 è usata comunemente per contare. Una forma unaria dell'istruzione `ADD` è l'operazione `INC` che incrementa di 1. L'operazione di negazione è un altro esempio: negare X corrisponde a calcolare $0 - X$, una sottrazione binaria, e poiché è un'operazione di uso frequente alle volte si fornisce un'apposita istruzione unaria `NEG`. Sottolineamo qui la differenza tra l'operazione aritmetica `NEG` e l'operazione logica `NOT`. L'operazione `NEG` produce l'**opposto** di un numero, quello cioè che dà 0 se sommato al numero originale. L'operazione `NOT` si limita a invertire i bit della parola. Le due operazioni sono molto simili, e infatti sono identiche nella rappresentazione in complemento a uno (nell'aritmetica in complemento a due, l'istruzione `NEG` si ottiene invertendo i singoli bit e poi aggiungendo 1).

Le istruzioni a uno o due operandi sono spesso raggruppate in base al loro uso invece che secondo il numero di operandi richiesto. Un insieme contiene le operazioni aritmetiche, compresa la negazione. Un altro insieme comprende le operazioni logiche e lo scorrimento, dal momento che vengono spesso usate congiuntamente per realizzare l'estrazione di dati.

5.5.4 Confronti e salti condizionati

Quasi tutti i programmi hanno bisogno della capacità di esaminare il contenuto dei dati e alterare la sequenza di esecuzione delle istruzioni in base al risultato dell'ispezione. Un semplice esempio di ciò è la funzione radice quadrata, $\sqrt{x}$: se x è negativo la procedura restituisce un messaggio d'errore, altrimenti calcola la radice quadrata. La funzione `sqrt` (da *square root*) deve perciò essere in grado di esaminare x e di scegliere come proseguire nell'esecuzione a seconda che x sia o meno negativa.

Un metodo comune per farlo è fornire istruzioni di salto condizionato che verificano una certa condizione e saltano a un particolare indirizzo di memoria se la condizione è soddisfatta. Alle volte c'è un bit nell'istruzione che indica se il salto deve avvenire a condizione soddisfatta o non soddisfatta. Spesso l'indirizzo di destinazione del salto non è assoluto, ma relativo all'istruzione corrente.

La condizione che viene testata più comunemente è se un determinato bit della macchina è posto o meno a 0. Se un'istruzione esamina il bit di segno di un numero e salta all'etichetta `ETICHETTA` qualora il bit valga 1, allora il comando che si trova alla posizione `ETICHETTA` sarà eseguito solo se il numero della condizione è negativo, altrimenti (se il numero è nullo o positivo) verrà eseguito il comando che segue il salto condizionato.

Molte macchine hanno bit usati per specificare certe condizioni. Per esempio ci potrebbe essere un bit di overflow asserito ogniqualvolta un'operazione aritmetica produce un risultato errato. Esaminare questo bit vuol dire verificare se l'istruzione precedente abbia o meno causato un overflow, e in caso positivo è possibile saltare a una routine di errore in grado di intraprendere le dovute azioni correttive.

Analogamente, alcuni processori hanno un bit di riporto che viene asserito quando si verifica un riporto all'ultimo bit di sinistra, per esempio quando vengono sommati due numeri negativi. Un riporto al bit più significativo è un evento normale, da non confondere con un overflow. Il test del bit di riporto è necessario all'aritmetica in precisione multipla (per esempio quando un intero è rappresentato in due o più parole).

Verificare se un parola vale zero è importante per i cicli e per molti altri scopi. Se tutte le istruzioni di salto condizionato esaminassero un bit alla volta, ci vorrebbero tanti confronti quanti sono i bit per verificare che tutti i bit di una parola valgono zero. Per evitare questo scenario, molte macchine mettono a disposizione istruzioni di confronto per esaminare una parola intera e, se nulla, effettuare il salto. Naturalmente questa soluzione non fa che passare la "patata bollente" alla microarchitettura. Di solito l'hardware contiene un registro i cui bit sono tutti messi in OR al fine di disporre di un solo bit che stabilisce immediatamente se la parola contiene uno o più bit asseriti. Il bit Z della Figura 4.1 viene calcolato di norma invertendo il risultato dell'OR di tutti i bit in uscita dalla ALU.

I confronti tra parole o tra caratteri sono importanti per verificare se sono uguali e, se non lo sono, per stabilire qual è maggiore, il che è indispensabile per poterli ordinare. Per effettuare questo test sono necessari tre indirizzi: due per i dati e uno verso cui saltare se la condizione è vera. I computer con formati d'istruzioni a tre indirizzi non incontrano difficoltà in tal senso, gli altri devono impiegare qualche espediente per aggirare il problema.

Una soluzione comune prevede un'istruzione che effettua il confronto e imposta uno o più bit di condizione per memorizzare il risultato. Un'istruzione successiva può esaminare i bit di condizione e saltare se i due valori messi a confronto si sono rivelati uguali,

o diversi, o se il primo era maggiore, e così via. È l'approccio prediletto dal Pentium 4 e dall'UltraSPARC III.

Il confronto di due numeri implica la comprensione di alcune sottigliezze. Infatti il confronto non è facile quanto una sottrazione: la sottrazione tra un numero positivo molto grande e uno negativo molto grande provoca un overflow, poiché il risultato della sottrazione non può essere rappresentato. L'istruzione di confronto invece deve stabilire se la condizione in esame è soddisfatta o meno e restituire sempre il risultato corretto; un confronto non può provocare un overflow.

Un'altra questione delicata nel confronto tra numeri è stabilire se un numero è da considerarsi con o senza segno. I numeri binari di tre bit possono essere ordinati in uno dei due modi. Dal più piccolo al più grande:

senza segno	con segno
000	100 (più piccolo)
001	101
010	110
011	111
100	000
101	001
110	010
111	011 (più grande)

La colonna di sinistra mostra gli interi positivi da 0 a 7 in ordine crescente. La colonna di destra mostra gli interi con segno in complemento a due che vanno da –4 a +3. Rispondere alla domanda "se 011 sia maggiore di 100" dipende dal modo in cui sono letti i numeri, se con segno o senza segno. La maggior parte degli ISA dispone d'istruzioni per gestire entrambi gli ordinamenti.

5.5.5 Invocazione di procedura

Una procedura è un insieme d'istruzioni che svolge un certo compito e che può essere invocata (chiamata) da diversi punti di un programma. Il termine **subroutine** è usato spesso invece di procedura, soprattutto in riferimento a programmi in linguaggio assemblativo. Il termine usato in Java è **metodo**. Quando una procedura ha terminato il proprio compito l'esecuzione deve riprendere dall'istruzione successiva alla chiamata. A tal fine l'indirizzo di ritorno deve essere passato alla procedura o salvato da qualche parte dove possa essere recuperato al momento del ritorno.

L'indirizzo di ritorno può essere salvato in tre posti diversi: in memoria, in un registro o nello stack. La soluzione di gran lunga peggiore è metterlo in una locazione di memoria fissa. In questo modo se la prodecura chiamasse un'altra procedura, la seconda chiamata causerebbe la perdita dell'indirizzo di ritorno della prima.

Un lieve miglioramento consiste nel far sì che l'istruzione di chiamata di procedura salvi l'indirizzo di ritorno nella prima parola della procedura, cosicché la prima istruzione del codice eseguibile si trovi nella seconda parola. La procedura può quindi rientrare saltando indirettamente tramite la prima parola o anche direttamente, se l'hardware consente l'inserimento dell'opcode per il salto direttamente nella prima parola, insieme all'indirizzo di ritorno. La procedura potrebbe richiamare altre procedure, visto che ciascuna alloca lo spazio

per il proprio indirizzo di ritorno. Questo schema fallisce però se la procedura chiama se stessa, perché il primo indirizzo di ritorno verrebbe cancellato dalla seconda chiamata. La capacità di una procedura di chiamare se stessa, detta **ricorsione**, è di estrema importanza tanto per gli studiosi di programmazione quanto per i programmatori. Questo schema fallisce anche se la procedura *A* chiama la procedura *B*, *B* chiama la procedura *C* e *C* chiama *A* (ricorsione indiretta o "a festone", *daisy-chain*).

Un miglioramento sostanziale si ottiene se l'istruzione di chiamata di procedura pone l'indirizzo di ritorno in un registro, affidando alla procedura la responsabilità di salvarlo in un posto sicuro. Se la procedura è ricorsiva dovrà preoccuparsi di salvare l'indirizzo di ritorno in un posto differente ogni volta che viene invocata.

La cosa migliore che può fare l'istruzione di chiamata di procedura è porre l'indirizzo di ritorno in cima a uno stack. Quando la procedura termina fa un pop dell'indirizzo di ritorno e lo scrive nel program counter. Con questa forma di chiamata la ricorsione non causa alcun problema particolare; l'indirizzo di ritorno viene salvato automaticamente senza sovrascrivere indirizzi di ritorno precedenti. In queste condizioni la ricorsione funziona perfettamente. Abbiamo già visto questa modalità di salvataggio dell'indirizzo di ritorno in IJVM (Figura 4.12).

5.5.6 Istruzioni di ciclo

Poiché capita spesso di dover eseguire un gruppo d'istruzioni un numero prefissato di volte, molte macchine dispongono d'istruzioni per facilitare questo compito. Tutti gli schemi prevedono un contatore che viene incrementato o decrementato di una certo valore costante a ogni iterazione del ciclo. Il contatore viene anche esaminato a ogni iterazione; il ciclo termina quando si verifica una certa condizione.

Un metodo possibile è inizializzare il contatore al di fuori del ciclo e quindi cominciarne immediatamente l'esecuzione. L'ultima istruzione del ciclo aggiorna il contatore e, se la condizione di terminazione non è stata ancora soddisfatta, torna alla prima istruzione del ciclo. In caso contrario il ciclo termina e si prosegue dalla prima istruzione dopo il ciclo. Questa tipologia di ciclo è detta con valutazione in coda. Nella Figura 5.30(a) ne diamo un esempio in C (non avremmo potuto scriverlo in Java perché non fornisce istruzioni di `goto`).

```
        i = 1;
L1:     Primalstruzione;
        .
        .
        .
        Ultimalstruzione;
        i = i + 1;
        if (i < n) goto L1;
```

(a)

```
        i = 1;
L1:     if (i > n) goto L2;
        Primalstruzione;
        .
        .
        .
        Ultimalstruzione;
        i = i + 1;
        goto L1;
L2:
```

(b)

Figura 5.30 (a) Ciclo con verifica della condizione in coda. (b) Ciclo con verifica della condizione in testa.

Il ciclo con valutazione in coda ha la proprietà di essere eseguito sempre almeno una volta, anche se n fosse minore a uguale a 0. Considerate a mo' d'esempio un programma gestore delle schede del personale di una ditta. A un certo punto della sua esecuzione il programma legge i dati di un particolare impiegato e salva in n il numero di figli che ha. Quindi esegue n volte un ciclo in cui, per ogni figlio, legge il suo nome, il sesso e la data di nascita affinché la ditta possa spedirgli un regalo di compleanno. Se l'impiegato non ha figli, n vale 0, eppure il ciclo verrà eseguito una volta e potrebbe spedire erroneamente un regalo di compleanno.

La Figura 5.30(b) mostra un'altra modalità di valutazione che funziona correttamente anche per valori di n minori o uguali a 0. Notate come il confronto cambi nelle due modalità, perciò se l'incremento e la valutazione sono effettuati da un'unica istruzione ISA, i progettisti sono costretti a scegliere una modalità o l'altra.

Considerate il codice che dovrebbe essere generato per l'istruzione

```
for (i = 0; i < n; i++) { istruzioni }
```

Se il compilatore non dispone d'informazioni su n, allora deve usare l'approccio della Figura 5.30(b) per poter gestire correttamente anche i casi in cui $n \leq 0$. Se invece può stabilire che $n > 0$, magari osservando la locazione cui n è stata assegnata, allora può usare il codice migliore della Figura 5.30(a). Il vecchio FORTRAN stabiliva che tutti i cicli dovevano essere eseguiti almeno una volta, proprio per consentire sempre la generazione del codice più efficiente della Figura 5.30(a). Nel 1977 anche la comunità FORTRAN capì che non era una buona idea avere un'istruzione di ciclo dalla semantica stravagante, e che ogni tanto restituiva una risposta errata, solo per risparmiare un'istruzione di salto per ciclo, e corresse il tiro. C e Java non hanno mai avuto questo problema.

5.5.7 Input/Output

Nessun raggruppamento d'istruzioni manifesta la stessa variabilità tra macchine diverse come le istruzioni di I/O. Attualmente i personal computer usano tre schemi diversi di I/O:

1. I/O programmato con attesa attiva
2. I/O *interrupt driven* (cioè innescato dagli interrupt)
3. I/O con DMA.

Vediamoli in dettaglio. Il metodo di I/O più semplice possibile è quello **programmato**, impiegato di solito nei microprocessori di fascia bassa come, per esempio, i sistemi integrati o i sistemi che devono rispondere rapidamente agli stimoli esterni (sistemi in tempo reale). Queste CPU hanno in genere una sola istruzione di input e una sola di output, che trasferisce un carattere per volta da un prefissato registro al dispositivo di I/O prescelto. Il processore deve eseguire una sequenza prestabilita d'istruzioni per ciascun carattere letto o scritto.

Come semplice esempio di questo metodo, si consideri il terminale con quattro registri di 1 byte mostrato nella Figura 5.31. Due registri sono utilizzati per lo stato e i dati in input, mentre gli altri due sono usati per lo stato e i dati in output. Ogni registro ha un indirizzo unico. Se l'I/O è mappato in memoria, i quattro registri fanno parte dello spazio degli indirizzi della memoria del computer e possono essere letti e scritti mediante le istruzioni ordinarie. In caso contrario ci sono istruzioni speciali di I/O, che chiamiamo `IN` e `OUT`, per

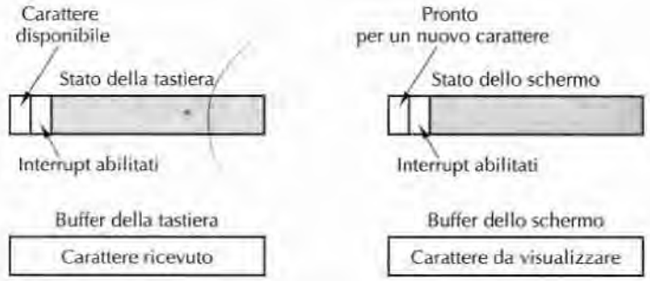


Figura 5.31 Registri dei dispositivi di un semplice terminale.

la lettura e la scrittura dei registri. In entrambi i casi l'I/O avviene tramite il trasferimento dati e del loro stato tra la CPU e questi registri.

Il registro dello stato della tastiera utilizza 2 soli bit degli 8 disponibili: il bit più significativo (il 7) è posto a 1 via hardware ogniqualvolta arriva un carattere; questo evento solleva un interrupt se e solo se il bit 6 era stato asserito precedentemente via software (tratteremo gli interrupt più avanti). Nel caso dell'I/O programmato la CPU aspetta dati in ingresso effettuando un ciclo serrato e ripetuto di letture sul registro di stato della tastiera, aspettando che il bit 7 assuma il valore 1. Non appena ciò si verifica, il software legge il carattere dal registro buffer della tastiera. La lettura del registro dati della tastiera causa l'azzeramento del bit `CARATTERE DISPONIBILE`.

L'output funziona in modo analogo. Per visualizzare un carattere sullo schermo il software per prima cosa legge il registro di stato dello schermo per assicurarsi che il bit `PRONTO` valga 1 e in caso negativo cicla finché il bit non viene asserito a indicare che il dispositivo è pronto ad accettare un carattere. Non appena il terminale è pronto, il software scrive un carattere nel registro di buffer dello schermo, il che provoca la sua trasmissione allo schermo e anche l'azzeramento del bit `PRONTO` (nel registro di stato dello schermo) da parte del dispositivo. Dopo la visualizzazione del carattere, il controllore imposta automaticamente a 1 il bit `PRONTO` non appena il dispositivo è preparato a trattare il carattere successivo.

Un esempio di I/O programmato è dato dalla procedura Java nella Figura 5.32. Questa procedura presenta due parametri: un array di caratteri per riporre il risultato e il numero di caratteri (*count*) presenti nell'array (al massimo 1024). Il corpo della procedura è costituito da un ciclo che restituisce un carattere alla volta. Per ogni carattere la CPU deve attendere innanzitutto che il dispositivo sia pronto prima di poter ottenerlo. È ragionevole pensare che le procedure *in* e *out* siano delle routine scritte in linguaggio assemblativo per la lettura e la scrittura dei registri di dispositivo specificati dai rispettivi parametri (il registro di stato per *in*, quello di buffer per *out*). La divisione per 128 permette di sbarazzarsi dei 7 bit meno significativi, riportando il bit `PRONTO` in posizione 0.

Lo svantaggio principale dell'I/O programmato è che la CPU passa gran parte del suo tempo in un ciclo serrato in cui attende che il dispositivo risulti pronto. Questo approccio si chiama **attesa attiva** (*busy waiting*) ed è accettabile se la CPU non ha null'altro da eseguire (sebbene anche un semplice controllore debba spesso monitorare molteplici eventi concorrenti). Tuttavia questa strategia è uno spreco in tutti quei casi in cui ci siano da svolgere

```
public static void output_buffer(char buf[ ],int count) {
  // Spedisce un blocco di dati al dispositivo
  int status, i, ready;
  for (i = 0; i < count; i++) {
    do {
      status = in(display_status_reg);          // ottiene lo stato
      ready = (status >> 7) & 0x01;             // isola il bit PRONTO
    } while (ready != 1);
    out(display_buffer_reg, buf[i];
  }
}
```

Figura 5.32 Esempio di I/O programmato.

altri compiti, quali l'esecuzione di ulteriori programmi, e richiede perciò l'individuazione di un altro metodo di I/O.

Un modo per evitare l'attesa attiva è far sì che la CPU faccia partire il dispositivo di I/O e gli impartisca l'ordine di generare un interrupt quando ha finito. Possiamo capire come funziona guardando la Figura 5.31. Il software può richiedere all'hardware di segnalargli quando un'operazione di I/O è conclusa tramite l'asserzione del bit INTERRUPT ABILITATI nel registro di dispositivo. Riprenderemo l'analisi degli interrupt più avanti in questo capitolo quando tratteremo il controllo del flusso.

Val la pena far presente che in molti computer il segnale di interrupt viene generato mettendo in AND i bit INTERRUPT ABILITATI e PRONTO. Se il software abilitasse gli interrupt prima di cominciare le operazioni di I/O, verrebbe sollevato immediatamente un interrupt, poiché il bit PRONTO varrebbe 1. Dunque potrebbe essere indispensabile prima far partire il dispositivo, per poi abilitare immediatamente gli interrupt. La scrittura di un byte nel registro di stato non modifica il bit PRONTO, che è di sola lettura.

Benché l'I/O guidato dagli interrupt costituisca un grande progresso rispetto all'I/O programmato, è ben lungi dall'essere perfetto. Il problema è che ci vuole un interrupt per ogni carattere trasferito, e l'elaborazione di un interrupt è gravosa. Occorre un modo per ridurre drasticamente il numero di interrupt.

La soluzione si ottiene tornando all'I/O programmato, ma affidandolo a qualcun altro che non sia la CPU (la soluzione di molti problemi consiste nell'affidare il lavoro a qualcun altro). La Figura 5.33 mostra come fare: aggiungiamo al sistema un nuovo chip, il controllore **DMA** (*Direct Memory Access*), con accesso diretto al bus.

Il chip DMA ha al suo interno almeno quattro registri accessibili dal software in esecuzione nella CPU: il primo contiene l'indirizzo di memoria di partenza per la lettura o scrittura; il secondo conta il numero di byte (o parole) da trasferire; il terzo specifica il numero di dispositivo o lo spazio d'indirizzamento di I/O da usare, il che determina il dispositivo di I/O desiderato; il quarto stabilisce se i dati vanno letti dal dispositivo di I/O o se vanno scritti su di esso.

Per trasferire un blocco di 32 byte dall'indirizzo di memoria 100 al terminale (sia questo il dispositivo 4) la CPU scrive i numeri 32, 100 e 4 nei primi tre registri DMA, più il codice per la scrittura (poniamo sia 1) nel quarto registro, come illustrato nella Figura 5.33. A questo punto il DMA effettua una richiesta di bus per leggere il byte 100 dalla memoria, analo-

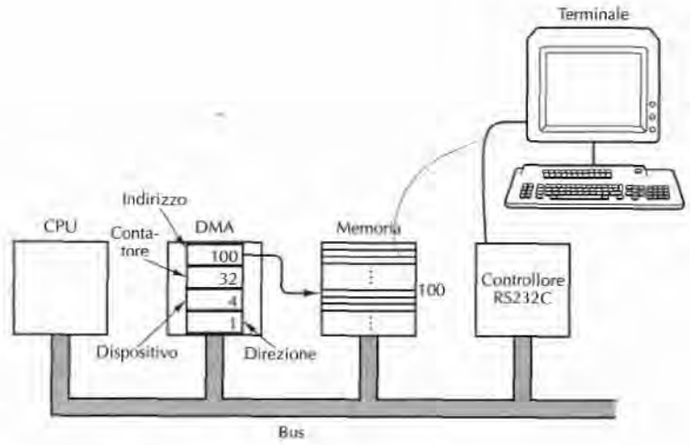


Figura 5.33 Controllore DMA.

gamente a come farebbe la CPU. Una volta ottenuto il byte, il controllore DMA effettuerebbe una richiesta di I/O al dispositivo 4 finalizzata alla scrittura del byte. Dopo il completamento di queste due operazioni, il controllore DMA incrementa di 1 il suo registro d'indirizzo e decrementa di 1 il suo registro contatore. Se il registro contatore è ancora positivo, si prosegue con la lettura da memoria di un altro byte e con la relativa scrittura nel dispositivo.

Infine, quando il contatore si azzera, il controllore DMA smette di trasferire dati e manda un impulso sulla linea di interrupt collegata al chip della CPU. In presenza di DMA, la CPU deve solo inizializzare pochi registri, dopo di che è libera di svolgere altri compiti fino al completamento del trasferimento, segnalato da un interrupt proveniente dal controllore DMA. Alcuni controllori DMA dispongono di due, tre o più insiemi di registri per controllare trasferimenti simultanei.

Anche se con il DMA la CPU viene sollevata dal carico pesante dell'I/O, il procedimento non è del tutto gratuito. Se un dispositivo ad alta velocità, per esempio un disco, è in fase di trasferimento controllato dal DMA, ci vorranno molti cicli di bus per gli accessi alla memoria e al dispositivo. Durante questi cicli la CPU deve restare in attesa (il DMA ha una priorità di bus sempre maggiore di quella della CPU, perché i dispositivi di I/O difficilmente tollerano i ritardi). Il fenomeno che si verifica quando il controllore DMA sottrae cicli di bus alla CPU si dice **appropriazione di cicli** (*cycle stealing*, "furto di cicli"). Nondimeno il guadagno che si ottiene nel non dover gestire un interrupt per byte (o per parola) ripaga largamente del danno derivato dall'appropriazione di cicli.

5.5.8 Istruzioni di Pentium 4

In questo paragrafo e nei due successivi analizzeremo gli insiemi d'istruzioni delle nostre tre architetture esemplificative: il Pentium 4, l'UltraSPARC e l'8051. Ciascuna di loro dispone di un nucleo d'istruzioni generate normalmente dai compilatori, più un insieme d'istruzioni usate raramente o destinate esclusivamente al sistema operativo. Nella nostra analisi focalizziamo l'attenzione sulle istruzioni comuni. Cominciamo dal Pentium 4.

L'insieme d'istruzioni del Pentium 4 è una miscellanea di vere istruzioni di 32 bit e resti dell'8088, suo antenato diretto. Nella Figura 5.34 mostriamo una piccola selezione di comuni istruzioni utilizzabili facilmente dai compilatori e dai programmatori odierni. È una lista tutt'altro che completa, visto che non include le istruzioni in virgola mobile, le istruzioni di controllo, né altre di uso meno frequente (tipo l'uso del byte `AL` come chiave di ricerca). In ogni caso restituisce una rappresentazione abbastanza fedele delle potenzialità del Pentium 4.

Molte delle istruzioni del Pentium 4 referenziano uno o due operandi, che possono trovarsi nei registri o in memoria. Per esempio l'istruzione `ADD` somma la sorgente alla destinazione, mentre la `INC` incrementa il suo operando di 1. Alcune istruzioni presentano varianti strettamente connesse tra loro. Per esempio l'istruzione di scorrimento può spostare i bit verso sinistra o verso destra, e può trattare il bit di segno in modo speciale o meno. Gran parte delle istruzioni presenta una varietà di codifiche, che cambia a seconda della natura degli operandi.

I campi `SRC` della Figura 5.34 sono sorgenti d'informazioni e non vengono modificati. Invece i campi `DST` sono destinazioni e di norma vengono modificati dall'istruzione. Ci sono alcune regole che stabiliscono che cosa può fungere da sorgente e che cosa da destinazione, che cambiano da istruzione a istruzione senza nessun criterio particolare, ma non ce ne curiamo in questa sede. Molte istruzioni presentano tre varianti, con operandi di 8, 16 e 32 bit. La distinzione tra i tipi avviene tramite l'opcode e/o un bit dell'istruzione. La lista della figura si sofferma soprattutto sulle istruzioni di 32 bit.

Per semplicità abbiamo suddiviso le istruzioni in vari gruppi: il primo contiene le istruzioni per il trasferimento dati all'interno della macchina, tra registri, e locazioni di memoria o di stack. Il secondo gruppo si occupa dell'aritmetica, con o senza segno. Nel caso della moltiplicazione e della divisione, il prodotto o il dividendo di 64 bit sono memorizzati in `EAX` (la parte meno significativa) e `EDX` (la parte più significativa).

Il terzo gruppo si occupa dell'aritmetica dei decimali in codifica binaria, detti anche BCD, che tratta ogni byte come due **nibble** (mezzi byte) di 4 bit. Ogni nibble contiene una cifra decimale (da 0 a 9); le configurazioni da 1010 a 1111 sono inutilizzate. Così un intero di 16 bit può contenere un numero decimale compreso tra 0 e 9999. Questa codifica è poco efficiente, ma non richiede la conversione dell'input da decimale in binario e quindi dell'output in decimale. Queste istruzioni sono usate per l'aritmetica dei numeri BCD, molto utilizzati dai programmi COBOL.

Le istruzioni booleane e quelle di scorrimento/rotazione manipolano in modi diversi i bit all'interno dei byte o delle parole. Ne presentiamo diverse varianti.

I due gruppi successivi trattano test, confronti, e salti basati sui loro risultati. I risultati dei test e dei confronti sono salvati in vari bit del registro `EFLAGS`. `Jxx` individua un insieme d'istruzioni di salto condizionato dal risultato del confronto precedente (cioè dai bit di `EFLAGS`).

Il Pentium 4 dispone di parecchie istruzioni per caricamento, memorizzazione, trasferimento, confronto e scansione di stringhe di caratteri o di parole. Queste istruzioni possono essere precedute da un byte speciale chiamato `REP` che ne causa l'esecuzione ripetuta finché non venga soddisfatta la condizione che `ECX`, decrementato a ogni iterazione, abbia raggiunto il valore 0. Così facendo è possibile trasferire, confrontare, e così via, blocchi arbitrari di dati. Il gruppo successivo gestisce i codici di condizione.

Trasferimenti

MOV DST,SRC	Trasferisce SRC in DST
PUSH SRC	Impila SRC sullo stack
POP DST	Pop di una parola dallo stack e la pone in DST
XCHG DS1, DS2	Scambia DS1 e DS2
LEA DST,SRC	Carica in DST l'indirizzo effettivo di SRC
CMOVcc DST,SRC	Trasferimento condizionato

Aritmetica

ADD DST, SRC	Somma SRC a DST
SUB DST, SRC	Sottrae SRC da DST
MUL SRC	Moltiplica EAX con SRC (senza segno)
IMUL SRC	Moltiplica EAX con SRC (con segno)
DIV SRC	Divide EDX:EAX per SRC (senza segno)
IDIV SRC	Divide EDX:EAX per SRC (con segno)
ADC DST,SRC	Somma SRC a DST e somma il bit di riporto
SBB DST,SRC	Sottrae SRC e il riporto da DST
INC DST	Incrementa DST di 1
DEC DST	Decrementa DST di 1
NEG DST	Nega DST (lo sottrae a 0)

Decimali in codifica binaria

DAA	Codifica decimale
DAS	Codifica decimale per la sottrazione
AAA	Codifica ASCII per l'addizione
AAS	Codifica ASCII per la sottrazione
AAM	Codifica ASCII per la moltiplicazione
AAD	Codifica ASCII per la divisione

Booleane

AND DST, SRC	AND di SRC e DST, risultato in DST
OR DST, SRC	OR di SRC e DST, risultato in DST
XOR DST, SRC	OR esclusivo di SRC e DST, risultato in DST
NOT DST	Rimpiazza DST con il suo complemento a uno

Scorrimento/rotazione

SAL/ SAR DST,#	Scorrimento di DST verso s/d di # bit
SHL/ SHR DST,#	Scorrimento logico di DST verso s/d di # bit
ROL/ROR DST,#	Rotazione di DST verso s/d di # bit
RCL/ RCR DST,#	Rotazione di DST attraverso il riporto di # bit

Test/confronto

TEST SRC1,SRC2	AND degli operandi, modifica EFLAGS
CMP SRC1,SRC2	Modifica EFLAGS secondo SRC1- SRC2

Trasferimento del controllo

JMP ADDR	Salta all'indirizzo ADDR
Jxx ADDR	Salti condizionati da EFLAGS
CALL ADDR	Chiama la procedura all'indirizzo ADDR
RET	Ritorno da procedura
IRET	Ritorno da interrupt
LOOPxx	Cicla finché la condizione non sia soddisfatta
INT n	Innesca un interrupt software
INTO	Innesca un interrupt se il bit di overflow = 1

Stringhe

LODS	Carica una stringa
STOS	Memorizza una stringa
MOVS	Muove una stringa
CMPS	Copia una stringa
SCAS	Scandisce una stringa

Codici di condizione

STC	Asserisce il bit di riporto in EFLAGS
CLC	Azzera il bit di riporto in EFLAGS
CMC	Complemento del bit di riporto in EFLAGS
STD	Asserisce il bit di direzione in EFLAGS
CLD	Azzera il bit di direzione in EFLAGS
STI	Asserisce il bit di interrupt in EFLAGS
CLI	Azzera il bit di interrupt in EFLAGS
PUSHFD	Impila EFLAGS sullo stack
POPFD	Fa la pop di EFLAGS dallo stack
LAHF	Carica AH da EFLAGS
SAHF	Memorizza AH in EFLAGS

Miscellanea

SWAP DST	Cambia ordinamento dei byte di DST
CWQ	Estende EAX a EDX: EAX per la divisione
CWDE	Estende il numero di 16 bit in AX a EAX
ENTER SIZE, LV	Crea record d'attivazione di SIZE byte
LEAVE	Disfa il record d'attivazione creato da ENTER
NOP	Operazione nulla
HLT	Terminazione
IN AL, PORT	Un byte in input da PORT ad AL
OUT PORT, AL	Un byte in output da AL a PORT
WAIT	Attende un interrupt

s/d = sinistra/destra
SRC = sorgente
DST = destinazione
# = numero di scorrimenti/rotazioni
LV = puntatore alle variabili locali

Figura 5.34 Alcune istruzioni intere di Pentium 4.

L'ultimo gruppo è una miscellanea d'istruzioni che non rientrano in nessuna delle categorie precedenti e comprende le conversioni, la gestione dei record d'attivazione dello stack, la terminazione della CPU e l'I/O.

Il Pentium 4 prevede molti **prefissi**, tra cui il summenzionato `REP`. Ognuno di questi prefissi è un byte speciale che può precedere la maggior parte delle istruzioni, analogamente a `WIDE` nell'IJVM. `REP` provoca la ripetizione dell'istruzione successiva finché `ECX` non raggiunga 0, come già detto. `REPZ` e `REPNZ` causano la ripetizione dell'istruzione che precedono fintanto che il codice di condizione `Z` non viene, rispettivamente, asserito o azzerato. `LOCK` riserva il bus per tutta la durata dell'esecuzione di un'istruzione, per consentire la sincronizzazione multiprocessore. Altri prefissi vengono usati per forzare un'istruzione in modalità di 16 bit o di 32 bit, il che non solo cambia la lunghezza degli operandi, ma ridefinisce completamente le modalità d'indirizzamento. Infine il Pentium 4 dispone di un complesso schema di segmentazione del codice, dei dati, dello stack e altro, un altro retaggio dell'8088. Sono previsti alcuni prefissi per forzare l'uso d'indirizzi di memoria in determinati segmenti, ma (fortunatamente) non sono oggetto della nostra analisi.

5.5.9 Istruzioni di UltraSPARC III

La Figura 5.35 elenca tutte le istruzioni in modalità utente che un compilatore per UltraSPARC III potrebbe generare. Non sono incluse nella lista le istruzioni in virgola mobile, le istruzioni di controllo (come quelle per la gestione della cache o il riavvio di sistema), le istruzioni che coinvolgono spazi degli indirizzi diversi da quello dell'utente e le istruzioni obsolete. L'insieme è sorprendentemente piccolo: UltraSPARC III è davvero un RISC. Le istruzioni `LOAD` e `STORE` sono ovvie, con versioni per 1, 2, 4 o 8 byte. Quando si carica un numero costituito da meno di 64 bit in un registro (di 64 bit), si può decidere se estenderlo con segno o con zero (ci sono istruzioni disponibili a entrambi gli scopi).

Il gruppo successivo riguarda l'aritmetica. Solo le istruzioni il cui nome contiene i caratteri `CC` possono impostare i bit dei codici di condizione `NZVC`. Sulle macchine CISC sono molte le istruzioni che modificano i codici di condizione, ma su quelle RISC ciò non è auspicabile perché riduce la libertà del compilatore di spostare le istruzioni nel tentativo di riempire gli intervalli di ritardo. Se l'ordine originale delle istruzioni è A … B … C, dove A imposta alcuni codici di condizione poi esaminati da B, il compilatore non può inserire C tra A e B nel caso C modificasse i codici di condizione. Per questa ragione molte istruzioni vengono fornite in duplice versione, di modo che il compilatore usi preferenzialmente la versione che non modifica i codici di condizione, a meno che abbia intenzione di esaminarne il contenuto in un momento successivo. La moltiplicazione e la divisione con e senza segno sono incluse nel corredo.

L'aritmetica con tag (contrassegno) prevede un formato speciale che utilizza numeri di 30 bit. Può essere usata da quei linguaggi, come Smalltalk o Prolog, in cui il tipo delle variabili non è noto al momento della compilazione e può cambiare durante l'esecuzione. Per i numeri con tag il compilatore può generare un'istruzione ADD con tag, e solo in fase di esecuzione la macchina determina se impiegare una ADD intera o in virgola mobile.

Il gruppo di scorrimento/rotazione contiene uno scorrimento verso sinistra e due verso destra, ciascuno nella versione di 32 bit e in quella estesa di 64 bit. `SLL` trasla tutti i 64 bit

Caricamento

LDSB ADDR,DST	Carica byte con segno (8 bit)
LDUB ADDR,DST	Carica byte senza segno (8 bit)
LDSH ADDR,DST	Carica mezza parola con segno (16 bit)
LDUH ADDR,DST	Carica mezza parola senza segno (16 bit)
LDSW ADDR,DST	Carica parola con segno (32 bit)
LDUW ADDR,DST	Carica parola senza segno (32 bit)
LDX ADDR,DST	Carica parola estesa (64 bit)

Memorizzazione

STB SRC,ADDR	Memorizza byte (8 bit)
STH SRC,ADDR	Memorizza mezza parola (16 bit)
STW SRC,ADDR	Memorizza parola (32 bit)
STX SRC,ADDR	Memorizza parola estesa (64 bit)

Aritmetica

ADD R1,S2,DST	Somma
ADDCC "	Somma e modifica di cc
ADDC "	Somma con riporto
ADDCCC "	Somma con riporto e modifica di cc
SUB R1,S2,DST	Sottrazione
SUBCC "	Sottrazione e modifica di cc
SUBC "	Sottrazione con riporto
SUBCCC "	Sottrazione con riporto e modifica di cc
MULX R1,S2,DST	Moltiplicazione
SDIVX R1,S2,DST	Divisione con segno
UDIVX R1,S2,DST	Divisione senza segno
TADCC R1,S2,DST	Somma con tag

Scorrimento/rotazione

SLL R1,S2,DST	Scorr. logico a sinistra (32 bit)
SLLX R1,S2,DST	Scorr. logico a sinistra esteso (64 bit)
SRL R1,S2,DST	Scorr. logico a destra (32 bit)
SRLX R1,S2,DST	Scorr. logico a destra esteso (64 bit)
SRA R1,S2,DST	Scorr. aritmetico a destra (32 bit)
SRAX R1,S2,DST	Scorr. aritmetico a destra esteso (64 bit)

Booleane

AND R1,S2,DST	AND
ANDCC "	AND e modifica di cc
ANDN "	NAND
ANDNCC "	NAND e modifica di cc
OR R1,S2,DST	OR
ORCC "	OR e modifica di cc
ORN "	NOR
ORNCC "	NOR e modifica di cc
XOR R1,S2,DST	XOR
XORCC "	XOR e modifica di cc
XNOR "	NOR ESCLUSIVO
XNORCC "	NOR ESCLUSIVO e modifica di cc

Trasferimento di controllo

BPcc ADDR	Salto con predizione
BPr SRC,ADDR	Salta in base al registro
CALL ADDR	Chiamata di procedura
RETURN ADDR	Ritorno da procedura
JMPL ADDR,DST	Salta e collega
SAVE R1,S2,DST	Avanza finestra di registri
RESTORE "	Ripristina finestra di registri
Tcc CC,TRAP#	Trap condizionata
PREFETCH FCN	Prefetch dei dati dalla memoria
LDSTUB ADDR,R	Load/store atomica
MEMBAR MASK	Barriera di memoria

Miscellanea

SETHI CON,DST	Imposta i bit dal 10 al 31
MOVcc CC,S2,DST	Trasferimento condizionato
MOVr R1,S2,DST	Trasferimento in base al registro
NOP	Operazione nulla
POPC S1,DST	Censimento
RDCCR V,DST	Lettura del registro codici di condizione
WRCCR R1,S2,V	Scrittura del registro codici di condizione
RDPC V,DST	Lettura del program counter

SRC = registro sorgente
DST = registro destinazione
R1 = registro sorgente
S2 = sorgente: registro o immediata
ADDR = indirizzo di memoria

TRAP# = numero della trap
FCN = codice di funzione
MASK = tipo di operazione
CON = costante
V = indicatore di registro

CC = insieme di codici di condizione
R = registro destinazione
cc = codice di condizione
r = LZ, LEZ, Z, NZ, GZ, GEZ
Scorr. = scorrimento

Figura 5.35 Principali istruzioni intere di UltraSPARC III.

per mantenere la compatibilità con le vecchie versioni del software. Gli scorrimenti sono usati soprattutto per la manipolazione di bit. Gran parte delle macchine CISC dispone di un numero ingente d'istruzioni di scorrimento e rotazione, quasi tutte completamente inutili. Pochi scrittori di compilatori passeranno le notti a piangerne la scomparsa!

Il gruppo d'istruzioni booleane è analogo al gruppo delle aritmetiche. Comprende le operazioni AND, OR, OR ESCLUSIVO, ANDN, ORN, and XNORN. Le ultime tre sono di dubbia utilità, però possono essere svolte in un solo ciclo e richiedono poco hardware supplementare; per questo motivo sono state incluse nel progetto. Anche i progettisti delle macchine RISC di tanto in tanto soccombono alle tentazioni.

Il gruppo successivo d'istruzioni contiene i trasferimenti di controllo. `BPcc` rappresenta l'insieme delle istruzioni che saltano al verificarsi di varie condizioni e inoltre specificano nell'istruzione stessa se il compilatore prevede che il salto verrà intrapreso o meno. `BPr` esamina un registro e salta condizionatamente alla condizione rilevata.

Sono previste due modalità di chiamata di procedura. L'istruzione `CALL` usa il formato 4 della Figura 5.15 con un offset di 30 bit relativo al `PC` (espresso in *parole*). Questo valore è sufficiente a raggiungere un'istruzione entro 2 GB dal chiamante, in entrambe le direzioni. L'istruzione `CALL` deposita l'indirizzo di ritorno in `R15`, che diventa `R31` dopo la chiamata.

L'altra modalità di chiamata di procedura si ha usando l'istruzione `JMPL`, che è di formato 1a o 1b, che consente di porre l'indirizzo di ritorno in un registro qualsiasi. Questa modalità è utile quando l'indirizzo di destinazione è calcolato durante l'esecuzione.

`SAVE` e `RESTORE` manipolano la finestra di registri e il puntatore allo stack. Entrambe fanno scattare una trap quando la finestra successiva (o precedente) non è disponibile.

L'ultimo gruppo contiene alcune istruzioni di vario genere. `SETHI` serve per ovviare all'impossibilità di tenere un operando immediato di 32 bit in un registro. Il modo di procedere è usare `SETHI` per impostare i bit da 10 a 31 e affidare quindi i bit rimanenti all'istruzione successiva grazie al formato immediato (1b).

Istruzione	Come simularla
MOV SRC,DST	OR di SRC con G0 e memorizza il risultato in DST
CMP SRC1,SRC2	SUBCC di SRC2 da SRC1 e memorizza il risultato in G0
TST SRC	ORCC di SRC con G0 e memorizza il risultato in G0
NOT DST	XNOR di DST con G0
NEG DST	SUB di DST da G0 e memorizza il risultato in DST
INCDST	ADD 1 a DST (operando immediato)
DECDST	SUB 1 da DST (operando immediato)
CLR DST	OR di G0 con se stesso e memorizza il risultato in DST
NOP	SETHI di G0 a 0
RET	JMPL %I7+8,%G0

Figura 5.36 Alcune istruzioni simulate da UltraSPARC III.

L'istruzione di censimento (*population count*) è un mistero. Conta il numero di bit 1 in una parola. Si vocifera che sia utile alla simulazione di esplosioni di bombe atomiche e che il National Laboratory di Los Alamos (un grande finanziatore) guardi di buon occhio le macchine che la mettono a disposizione. Le ultime tre istruzioni servono alla lettura e scrittura di registri speciali.

Un gran numero di comuni istruzioni CISC manca dalla lista; può però essere simulato facilmente usando `G0` o un operando costante (formato 1b). Alcune di queste sono elencate nella Figura 5.36. Sono tutte riconosciute dall'assembler dell'UltraSPARC III e generate spesso dai compilatori. Molte di loro sfruttano il fatto che `G0` è cablato con il valore 0 e che istruzioni di scrittura su di esso non hanno alcun effetto.

5.5.10 Istruzioni di 8051

Questo processore ha un repertorio d'istruzioni semplice, la Figura 5.37 ne mostra la prima parte. Ogni riga specifica il nome abbreviato dell'istruzione, ne dà una descrizione concisa e indica le modalità d'indirizzamento per la sorgente o per la destinazione, a seconda che nella descrizione si parli di *src* o di *dst*. Come ci si poteva attendere, ci sono molte istruzioni `MOV` per il trasferimento di dati tra `ACC` (accumulatore), i registri e la memoria. Ci sono istruzioni per le operazioni di push e di pop su di uno stack indirizzato da un registro dedicato. Le locazioni di memoria oltre l'indirizzo 256 si trovano sempre nella memoria esterna, dato che l'8051 ha soli 128 byte di memoria e l'8052 ne ha appunto 256, e sono indirizzate sempre in modo indiretto tramite il registro di 16 bit `DPTR`. Il gruppo dei trasferimenti è completato da istruzioni miste per lo scambio di porzioni di registri.

L'8051 dispone di semplici istruzioni aritmetiche per somma, sottrazione, moltiplicazione e divisione, dove le ultime due usano registri prefissati. Sono disponibili anche istruzioni d'incremento e decremento, usate di frequente. Ci sono anche le istruzioni booleane e di rotazione.

Le restanti istruzioni dell'8051, elencate nella Figura 5.38, comprendono le istruzioni sui bit come, per esempio

```
SETB 43
```

che asserisce il bit 43, ma non modifica gli altri bit dello stesso byte. Vengono poi le istruzioni di trasferimento del controllo, compresi i salti e le chiamate di subroutine, più due salti condizionati `CJNE` che confrontano la sorgente con qualcos'altro, e l'istruzione `DJNZ` usata nei cicli.

5.5.11 Insiemi d'istruzioni a confronto

I tre esempi di insiemi d'istruzioni che abbiamo visto sono molto diversi tra loro. Il Pentium 4 è una classica macchina CISC a 32 bit e a due indirizzi, con una lunga storia, modalità d'indirizzamento peculiari e molto irregolari, nonché con molte istruzioni per l'accesso in memoria. L'UltraSPARC III è una moderna macchina RISC a 64 bit e a tre indirizzi, con architettura load/store, a malapena qualche modalità d'indirizzamento e un insieme d'istruzioni efficiente e compatto. L'8051 è un minuscolo processore integrato, progettato per essere realizzato su un solo chip.

Istr.	Descrizione	ACC	A reg.	Dir.	A reg. ind.	Imm.	C	bit
MOV	Trasf. *src* in ACC		×	×	×	×		
MOV	Trasf. *src* in un registro	×		×		×		
MOV	Trasf. *src* in memoria	×	×	×	×	×		
MOV	Trasf. *src* nella RAM indiretta	×		×		×		
MOV	Trasf. costante di 16 bit in DPTR							
MOVC	Trasf. codice all'offset in ACC relativo a DPTR							
MOVC	Trasf. codice all'offset in ACC relativo a PC							
MOVX	Trasf. byte da RAM esterna in ACC				×			
MOVX	Trasf. byte da RAM esterna in ACC tramite DPTR							
MOVX	Trasf. byte in RAM esterna da ACC				×			
MOVX	Trasf. byte in RAM esterna da ACC tramite DPTR							
PUSH	Push del byte *src* sullo stack			×				
POP	Pop del byte sullo stack in dst			×				
XCH	Scambia ACC e dst	×		×	×			
XCHD	Scambia delle cifre meno significative tra ACC e dst			×				
SWAP	Scambio incrociato dei nibble di dst	×						
ADD	Somma *src* ad ACC		×	×	×	×		
ADDC	Somma *src* ad ACC con riporto		×	×	×	×		
SUBB	Sottrae *src* da ACC		×	×	×	×		
INC	Incrementa dst	×	×	×	×			
DEC	Decrementa dst	×	×	×	×			
INC	DPTR							
MUL	Moltiplicazione							
DIV	Divisione							
DA	Codifica decimale di dst	×						
ANL	AND di *src* e ACC		×	×	×	×		
ANL	AND di ACC e dst			×				
ANL	AND di op. immediato e dst			×				
ORL	OR di *src* e ACC		×	×	×	×		
ORL	OR di ACC e dst			×				
ORL	OR di op. immediato e dst			×				
XRL	XOR di *src* e ACC		×	×	×	×		
XRL	XOR di ACC e dst			×				
XRL	XOR di op. immediato e dst			×				
CLR	Azzera dst	×						
CPL	Complementa dst	×						
RL	Rotazione di dst verso sinistra	×						
RLC	Rotazione di dst verso sinistra attraverso il riporto	×						
RR	Rotazione di dst verso destra	×						
RRC	Rotazione di dst verso destra attraverso il riporto	×						

Figura 5.37 Prima parte dell'insieme d'istruzioni di 8051.

Istr.	Descrizione	ACC	A reg.	Dir.	A reg. ind.	Imm.	C	bit
CLR	Azzera il bit						×	×
SETB	Asserisce il bit						×	×
CPL	Complementa il bit						×	×
ANL	AND di *src* e del riporto							×
ANL	AND del complemento di *src* e del riporto							×
ORL	OR di *src* e del riporto							×
ORL	OR del complemento di *src* e del riporto							×
MOV	Trasferisce *src* nel riporto							×
MOV	Trasferisce il riporto in *src*							×
JV	Salto relativo se il riporto è asserito							
JNC	Salto relativo se il riporto non è asserito							
JB	Salto relativo se il bit diretto è asserito							×
JNB	Salto relativo se il bit diretto non è asserito							×
JBC	Salto relativo se il bit diretto è asserito e il riporto no							×
ACALL	Chiamata di subroutine (indirizzo di 11 bit)							
LCALL	Chiamata di subroutine (indirizzo di 16 bit)							
RET	Ritorno da subroutine							
RETI	Ritorno da interrupt							
SJMP	Salto relativo breve (indirizzo di 8 bit)							
AJMP	Salto assoluto (indirizzo di 11 bit)							
LJMP	Salto assoluto (indirizzo di 16 bit)							
JMP	Salto indiretto relativo a DPR + ACC							
JZ	Salta se ACC è zero							
JNZ	Salta se ACC è diverso da zero							
CJNE	Confronta *src* con ACC, salta se diversi			×		×		
CJNE	Confronta *src* con op. immediato, salta se diversi		×		×			
DJNZ	Decrementa dst e salta se non nullo							
NOP	Operazione nulla							

Figura 5.38 Seconda parte dell'insieme d'istruzioni dell'8051.

Se ogni macchina è fatta a modo proprio ci sono delle buone ragioni. Il progetto del Pentium 4 risponde a tre esigenze principali:

1. retrocompatibilità
2. retrocompatibilità
3. retrocompatibilità.

Considerando lo stato dell'arte attuale, a nessuno verrebbe in mente di progettare una macchina così irregolare e con così pochi registri, tutti diversi tra loro. Queste caratteristiche complicano la scrittura dei compilatori. La mancanza di registri inoltre costringe i compilatori a riversare continuamente le variabili in memoria per poi ricaricarle, un lavoro gravoso an-

che se si hanno due o tre livelli di cache. La velocità del Pentium 4 è una riprova delle capacità degli ingegneri dell'Intel, a dispetto dei vincoli posti dall'ISA. Eppure abbiamo visto nel capitolo precedente quanto sia complessa la sua implementazione.

L'UltraSPARC III rappresenta un progetto di ISA allo stato dell'arte. Ha un ISA di 64 bit (con bus di 128 bit), ha molti registri e un insieme d'istruzioni che predilige le operazioni a tre registri, più un piccolo gruppo d'istruzioni LOAD e STORE. Tutte le istruzioni sono della stessa lunghezza, anche se il numero di formati è sfuggito un po' di mano. Ciononostante l'implementazione resta semplice ed efficiente. Gran parte dei nuovi progetti tende a somigliare all'UltraSPARC III, ma con meno formati d'istruzione.

L'8051 offre un insieme d'istruzioni semplice e abbastanza regolare, con relativamente poche istruzioni e poche modalità d'indirizzamento. Si distingue per la presenza di quattro insiemi di registri per l'eleborazione rapida degli interrupt, per la capacità di rendere i registri accessibili nello spazio di memoria e per le istruzioni di manipolazione dei bit sorprendentemente potenti. Il suo vanto principale è di richiedere un numero molto limitato di transitor per l'implementazione, il che consente di ospitarne un gran numero su di una sola piastra, dando luogo a un costo unitario di CPU molto contenuto.

5.6 Controllo del flusso

Il controllo del flusso riguarda la sequenza con cui le istruzioni vengono eseguite dinamicamente, ovvero durante l'esecuzione del programma. In mancanza d'istruzioni di salto o di chiamate di procedura, le istruzioni vengono eseguite di norma nello stesso ordine con cui si susseguono in memoria. Le chiamate di procedura alterano il controllo del flusso, perché arrestano la procedura correntemente in esecuzione e cominciano l'esecuzione della procedura chiamata. Le coroutine sono legate alle procedure e causano un'alterazione simile (sono utili nella simulazione di processi paralleli). Anche le trap e gli interrupt provocano un'alterazione del flusso esecutivo quando si verificano certe condizioni speciali. Sono tutti argomenti trattati nei paragrafi successivi.

5.6.1 Flusso sequenziale e diramazioni

Molte istruzioni non alterano il controllo del flusso; dopo l'esecuzione di un'istruzione, viene recuperata ed eseguita l'istruzione che la segue in memoria. Al termine di ciascuna istruzione, il program counter viene incrementato della lunghezza dell'istruzione elaborata. Se lo si osserva per un tempo abbastanza lungo, rispetto al tempo di esecuzione di una singola istruzione, il program counter è una funzione lineare nella variabile tempo e che aumenta, in ogni unità di tempo, di una quantità pari alla lunghezza media delle istruzioni diviso il loro tempo medio di esecuzione. In altre parole, l'ordine dinamico secondo cui il processore esegue le istruzioni è quello con cui appaiono nel listato del programma, come mostrato dalla Figura 5.39(a). Se un programma contiene salti, questa semplice relazione tra ordinamento delle istruzioni in memoria e ordine di esecuzione non vale più. In presenza di salti il program counter non è più una funzione monotòna crescente nel tempo, come si evince dalla Figura 5.39(b). In ragione di ciò, diventa difficile visualizzare la sequenza di esecuzione delle istruzioni a partire dal listato.

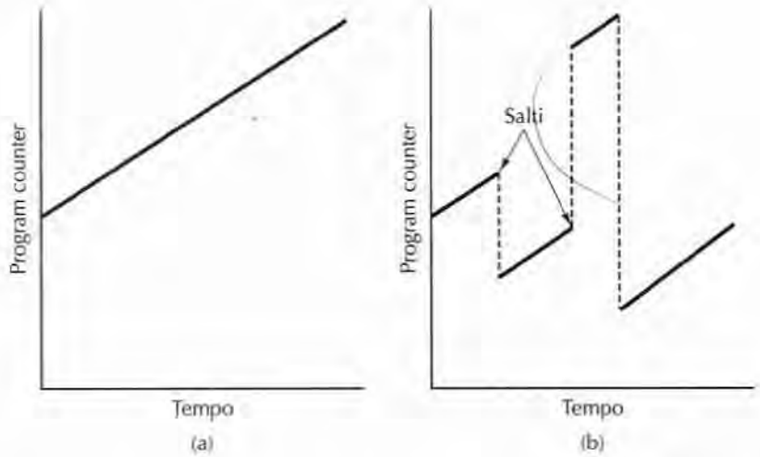


Figura 5.39 Valore del program counter in funzione del tempo. (a) Senza salti. (b) Con salti.

Quando i programmatori incontrano difficoltà nel tener traccia della sequenza di esecuzione delle istruzioni da parte del processore, commettono errori con più facilità. Questa osservazione ha portato Dijkstra a scrivere una nota (1968a), allora controversa, dal titolo "GO TO Statement Considered Harmful" ("L'istruzione GOTO è da considerarsi dannosa"), in cui suggeriva di evitare le istruzioni di `goto`. Da quel testo è scaturita la rivoluzione della programmazione strutturata, tra i cui dogmi c'è la sostituzione di tutti i comandi `goto` con forme di controllo più strutturate, quali i cicli. Naturalmente quando questi programmi vengono compilati in programmi di livello 2, questi ultimi possono contenere molti salti, dal momento che l'implementazione di `if`, `while` e di altre strutture di controllo di alto livello richiede la capacità di saltare da una parte all'altra.

5.6.2 Procedure

Le procedure costituiscono la tecnica più importante per la strutturazione dei programmi. Da un certo punto di vista una chiamata di procedura altera il flusso esecutivo tanto quanto un salto, ma a differenza di un salto alla terminazione del suo compito la procedura ripassa il controllo al comando o all'istruzione che segue la sua chiamata.

Tuttavia, da un altro punto di vista, il corpo di una procedura può essere visto come la definizione di una nuova istruzione di alto livello. Così facendo, la chiamata di procedura è concepibile come un'istruzione singola, senza riguardo alla sua effettiva complessità. Per la comprensione di una porzione di codice contenente una chiamata di procedura è sufficiente sapere *che cosa* fa, non *come* lo fa.

Una tipologia di procedure particolarmente interessanti è quella delle **procedure ricorsive**, ovvero quelle procedure che richiamano se stesse direttamente o indirettamente attraverso una successione di altre procedure. Lo studio delle procedure ricorsive favorisce una profonda comprensione del modo in cui sono implementate le chiamate di procedura, e della natura delle variabili locali. Diamo ora un esempio di procedura ricorsiva.

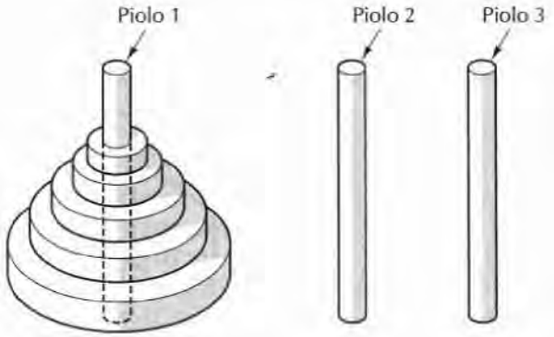


Figura 5.40 Configurazione iniziale del problema della torre di Hanoi con cinque dischi.

Quello della "torre di Hanoi" è un vecchio problema che ammette una semplice soluzione basata sulla ricorsione. In un monastero di Hanoi ci sono tre pioli d'oro: sul primo sono impilati, attraverso un foro apposito, 64 dischi d'oro. I dischi sono di dimensioni diverse e ciascuno è poggiato su di un disco di diametro maggiore del proprio. Gli altri due pioli sono inizialmente vuoti. I monaci del monastero hanno il compito di trasferire tutti i dischi (uno alla volta) sul terzo piolo, ma senza mai poggiare un disco di dimensioni maggiori su uno più piccolo. Si dice che quando avranno evaso il loro compito, il mondo avrà fine. Se volete sperimentare direttamente potete provare a usare dei dischi qualunque (e meno di 64), ma una volta che avrete risolto il problema non aspettatevi che succeda alcunché. Per ottenere l'effetto "fine del mondo" dovete usarne 64 e tutti d'oro. La Figura 5.40 mostra la configurazione iniziale con 5 dischi.

La soluzione consiste nel muovere gli n dischi dal piolo 1 al piolo 3 spostandone prima $n - 1$ dal piolo 1 al 2, quindi impilando il disco rimanente sul piolo 3 e infine spostando gli $n - 1$ dischi del piolo 2 sul piolo 3. La Figura 5.41 illustra questa soluzione (per $n = 3$).

Per risolvere il problema abbiamo bisogno di una procedura per spostare n dischi dal piolo i al piolo j. La sua invocazione

```
towers(n, i, j)
```

produce la visualizzazione della soluzione. Per prima cosa la procedura effettua il test $n = 1$. Se così fosse la soluzione sarebbe banale, basterebe spostare il disco da i a j. Se $n \neq 1$ allora la soluzione consiste nelle tre fasi già illustrate, ciascuna delle quali comporta una chiamata di procedura ricorsiva.

La soluzione completa è data dalla Figura 5.42. La chiamata

```
towers(3, 1, 3)
```

che risolve il problema della Figura 5.41, provoca altre tre chiamate, e cioè

```
towers(2, 1, 2)
towers(1, 1, 3)
towers(2, 2, 3)
```

La prima e la terza causano a loro volta tre chiamate, per un totale di sette.

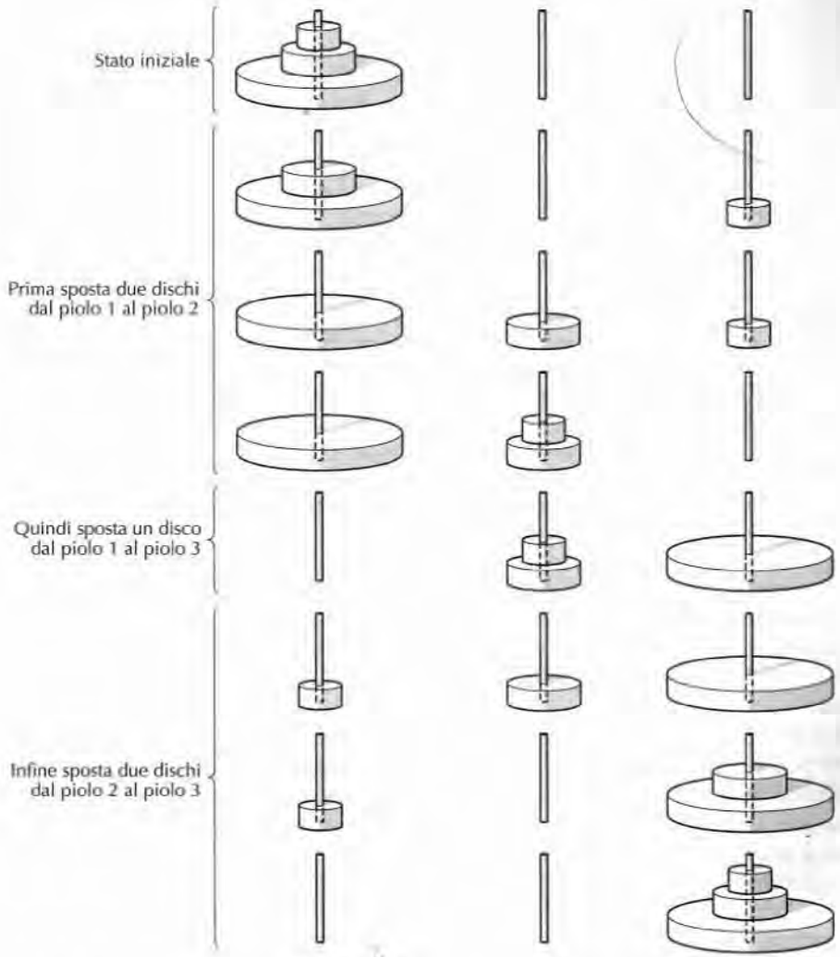


Figura 5.41 Passi necessari per la soluzione del problema della torre di Hanoi con tre dischi.

Al fine di poter gestire le procedure ricorsive abbiamo bisogno di uno stack per memorizzare i parametri e le variabili locali a ogni invocazione, analogamente a quanto mostrato nel caso dell'IJVM. Ogni volta che una procedura viene chiamata, in cima allo stack viene allocato un record d'attivazione per la procedura stessa. Il record d'attivazione di creazione più recente è quello in uso corrente. Nei nostri esempi lo stack cresce verso l'alto, dagli indirizzi di memoria più piccoli ai più grandi, proprio come nell'IJVM. Perciò il record d'attivazione più recente è caratterizzato da un indirizzo maggiore di tutti gli altri.

Oltre al puntatore allo stack, che punta alla cima della pila, risulta spesso conveniente avere un puntatore al record d'attivazione, FP (*frame pointer*), che punta a una locazione

```
public void towers(int n, int i, int j) {
  int k;

  if (n == 1)
    System.out.println( "Sposta un disco da " + i + " a " + j);
  else {
    k = 6 – i – j;
    towers(n – 1, i, k);
    towers(1, i, j);
    towers(n – 1, k, j);
  }
}
```

Figura 5.42 *Procedura per la soluzione del problema della torre di Hanoi.*

nota del record d'attivazione (per esempio il puntatore di collegamento, come nell'IJVM, o la prima variabile locale). La Figura 5.43 mostra il record d'attivazione di una macchina con parole di 32 bit. La chiamata originale `towers` impila *n*, *i* e *j* in cima allo stack ed esegue quindi un'istruzione `CALL` che impila l'indirizzo di ritorno sullo stack, all'indirizzo 1012. Al momento dell'ingresso, la procedura memorizza nello stack il vecchio valore di `FP` alla locazione 1016 e incrementa il puntatore allo stack per allocare spazio sufficiente alle variabili locali. Poiché c'è una sola variabile locale di 32 bit (*k*), `SP` è incrementato di 4 in 4 fino a 1020. La Figura 5.43(a) mostra la situazione dello stack al termine di queste operazioni.

Le prime cose che una procedura invocata deve fare sono il salvataggio del vecchio `FP` (così che possa essere ripristinato all'uscita della procedura), la copia di `SP` in `FP` e l'eventuale incremento di `FP` di una parola, a seconda della locazione cui puntare nel nuovo record d'attivazione. In questo esempio `FP` punta alla prima variabile locale, ma nell'IJVM `LV` indirizzava il puntatore di collegamento. La gestione del puntatore al record d'attivazione può essere leggermente diversa su macchine distinte, a seconda che abbia l'indirizzo della base del record d'attivazione, come nella Figura 5.43, o della sua cima oppure di altro. A tal fine è utile confrontare le Figure 5.43 e 4.12 per comprendere due modi alternativi di gestire il puntatore di collegamento. Ci sono anche altri modi possibili. Comunque sia, il punto chiave è poter effettuare un ritorno da procedura e ripristinare lo stato dello stack alla situazione in cui si trovava appena prima dell'invocazione della procedura corrente.

Il codice che si preoccupa di salvare il vecchio `FP`, che stabilisce il nuovo `FP` e accresce il puntatore allo stack per riservare spazio alle variabili locali si chiama **prologo della procedura** (*procedure prolog*). All'uscita dalla procedura si rende invece necessario ripulire lo stack, che rappresenta la fase di **epilogo della procedura**. Due delle caratteristiche più importanti di ogni computer sono la brevità e la velocità dei meccanismi di prologo ed epilogo delle procedure. Se sono lunghi e lenti, le chiamate di procedura ne soffriranno e i programmatori devoti all'efficienza impareranno a evitare di scrivere molte procedure brevi, cui preferiranno programmi lunghi, monolitici e poco strutturati. Le istruzioni `ENTER` e `LEAVE` del Pentium 4 sono state progettate proprio per far funzionare i prologhi e gli epiloghi delle procedure in modo efficiente. Naturalmente queste istruzioni hanno un loro modello di gestione del puntatore al record d'attivazione, per cui se il compilatore ha un modello diverso deve fare a meno di usarle.

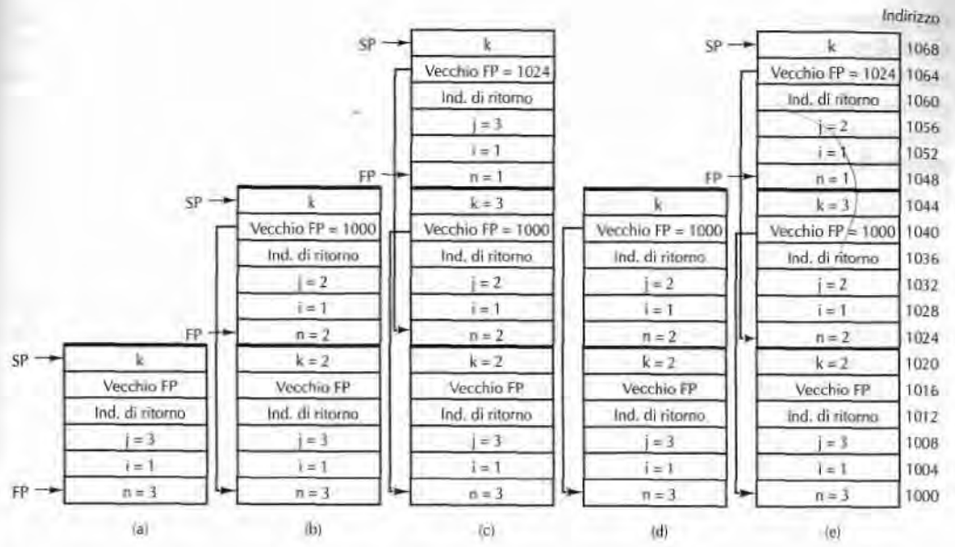


Figura 5.43 *Lo stack in alcuni momenti dell'esecuzione del codice della Figura 5.42.*

Torniamo ora al problema della torre di Hanoi. Ogni chiamata di procedura aggiunge un nuovo record d'attivazione allo stack che verrà rimosso al suo ritorno. Per illustrare l'uso dello stack nell'implementazione delle procedure ricorsive, tracciamo le chiamate a partire da

```
towers(3, 1, 3)
```

La Figura 5.43(a) mostra lo stack appena prima della chiamata di questa procedura. Per prima cosa la procedura controlla il valore di *n*, e quando scopre che $n = 3$, assegna *k* e opera la chiamata

```
towers(2, 1, 2)
```

Al completamento di questa chiamata lo stack è quello della Figura 5.43(b) e la procedura ricomincia dall'inizio (a ogni chiamata di procedura l'esecuzione riparte dall'inizio). Anche questa volta il confronto *n* è diverso da 1, perciò la procedura inizializza il proprio *k* e chiama

```
towers(1, 1, 3)
```

Ora lo stack è nella situazione della Figura 5.43(c) e il program counter punta all'inizio della procedura. Questa volta però $n = 1$ e viene visualizzata una riga di risultato. Dopodiché la procedura rientra e rimuove un record d'attivazione, ripristinando `FP` e `SP` come indicato nella Figura 5.43(d). Successivamente l'esecuzione continua dall'indirizzo di ritorno, che è la seconda chiamata:

```
towers(1, 1, 2)
```

Questa provoca l'aggiunta di un nuovo record allo stack come mostrato nella Figura 5.43(e). Quindi viene visualizzata una nuova riga di risultato e il record d'attivazione è rimosso dallo stack. Le chiamate di procedura proseguono con questo andamento, finché non viene completata l'esecuzione della chiamata originaria, al che viene rimosso dallo stack anche il record d'attivazione della Figura 5.43(a). Per capire davvero come funziona la ricorsione consigliamo di simulare con carta e penna l'intera esecuzione di

```
towers(3, 1, 3)
```

5.6.3 Coroutine

Nell'usuale sequenza di chiamata c'è una chiara distinzione tra procedura chiamante e procedura chiamata.Si consideri la Figura 5.44, dove la procedura *A* chiama la procedura *B*.

La procedura *B* svolge dei calcoli, e restituisce il controllo all'altra. A prima vista sembrerebbe una situazione simmetrica, perché né *A* né *B* costituiscono il programma principale, ma due procedure (la procedura *A* potrebbe essere stata invocata dal programma principale, ma ciò è irrilevante). Il controllo passa da *A* a *B*, durante la chiamata, mentre durante il rientro è trasferito da *B* ad *A*.

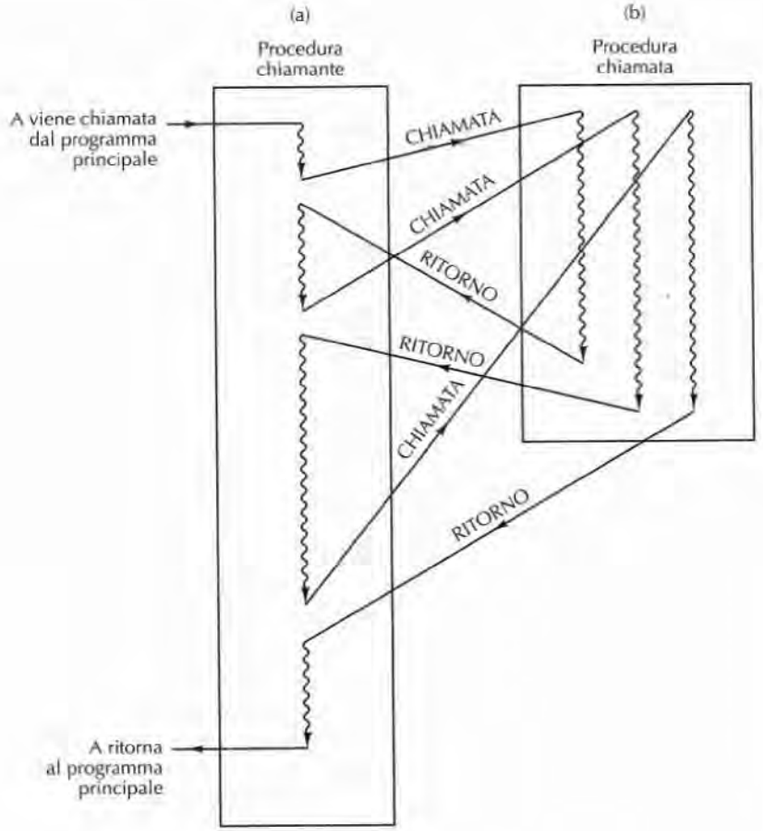


Figura 5.44 L'esecuzione di una procedura invocata inizia sempre dalla sua prima istruzione.

L'asimmetria nasce dal fatto che, nel primo caso l'esecuzione della procedura *B* comincia dal suo inizio, mentre nel secondo caso l'esecuzione non comincia dall'inizio di *A*, ma dall'istruzione immediatamente successiva alla chiamata. Se poi *A* chiama ancora *B*, l'esecuzione riparte dall'inizio di *B*, non dall'istruzione che segue il ritorno precedente. Ogniqualvolta *A* chiama *B* quest'ultima ricomincia dall'inizio, mentre *A* non ricomincia mai da capo.

Questa differenza si riflette nel modo in cui il controllo è passato tra *A* e *B*. Per chiamare *B*, la procedura *A* si avvale dell'istruzione di chiamata di procedura, che memorizza l'indirizzo di ritorno (cioè l'indirizzo dell'istruzione che segue la chiamata) in un qualche luogo adatto allo scopo, per esempio in cima allo stack. Quindi pone l'indirizzo di *B* nel program counter, completando così la chiamata. Quando *B* ritorna, non usa l'istruzione di chiamata ma si avvale dell'istruzione di ritorno, che semplicemente fa il pop dell'indirizzo di ritorno dallo stack e lo copia nel program counter.

Alle volte è utile avere due procedure che si chiamano a vicenda nel modo illustrato nella Figura 5.45. Come in precedenza, quando *B* ritorna ad *A*, *B* salta all'istruzione appena successiva alla sua chiamata. Quando è invece *A* a passare il controllo a *B*, non salta all'inizio di *B* (tranne la prima volta) bensì all'istruzione successiva al "ritorno" più recente, vale

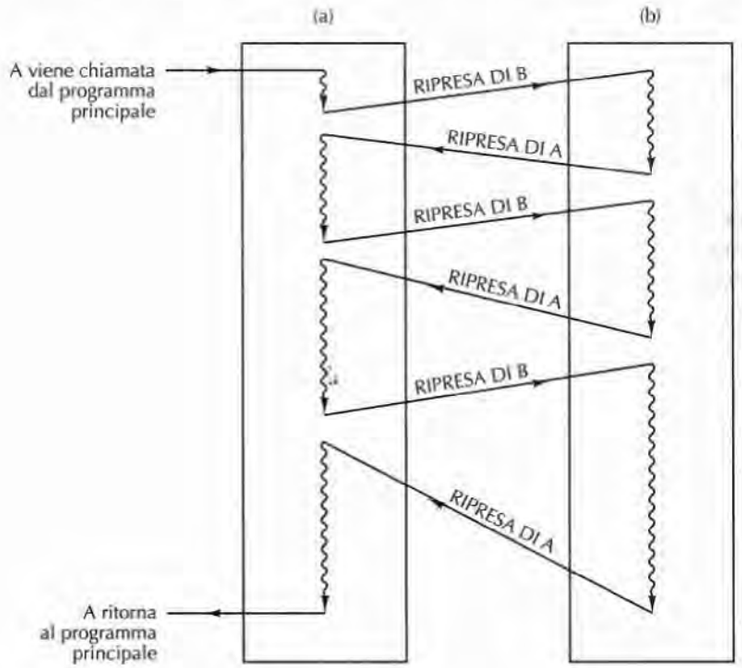


Figura 5.45 Alla ripresa di una coroutine, la sua esecuzione comincia da dove era stata interrotta, non dall'inizio.

a dire successiva alla chiamata di *A* più recente. Due procedure che si comportano in questo modo prendono il nome di **coroutine**.

Le coroutine sono usate comunemente per la simulazione dell'elaborazione parallela su singola CPU. Ogni coroutine gira in maniera pseudo-parallela alle altre, come se disponesse di una propria CPU. Questo stile di programmazione semplifica la scrittura di alcune applicazioni; inoltre è utile per la validazione del software destinato a girare su sistemi multiprocessore.

Le istruzioni `CALL` e `RETURN` non funzionano per le chiamate di coroutine perché, anche se l'indirizzo del salto viene recuperato dallo stack come per un'istruzione di ritorno, in questo caso è la stessa chiamata di coroutine a salvare l'indirizzo di ritorno per un successivo ritorno a essa. Sarebbe bello se esistesse un'istruzione per scambiare la cima dello stack con il program counter. In particolare questa istruzione effettuerebbe per prima cosa il pop del vecchio indirizzo di ritorno dallo stack verso un registro interno, quindi eseguirebbe una push del program counter sullo stack e infine copierebbe il contenuto del registro interno nel program counter. Dal momento che verrebbero effettuate sullo stack una push e un pop di una parola ciascuna, il puntatore allo stack non si sposterebbe. Questa istruzione si riscontra di rado, perciò viene spesso simulata per mezzo di numerose istruzioni.

5.6.4 Trap

Una **trap** ("trappola") è una specie di chiamata di procedura automatica effettuata quando si verificano certe condizioni causate da un programma e che sono in genere eventi rilevanti, ma di rara occorrenza. Ne è un buon esempio l'overflow: in molti computer, se il risultato di un'operazione aritmetica eccede il più grande numero rappresentabile, si verifica una trap, ovvero il controllo del flusso viene interrotto e riprende da una locazione di memoria prefissata, invece di proseguire in sequenza. In tale locazione di memoria si trova l'indirizzo per un salto a una procedura detta **gestore di trap**, che svolge le azioni appropriate al caso, come la stampa di un messaggio di errore. Viceversa non si verifica nessuna trap se il risultato di un'operazione è nell'intervallo dei numeri rappresentati dalla macchina.

Il concetto chiave delle trap è che sono fatte scattare da condizioni eccezionali causate dal programma stesso e rilevate dall'hardware o dal microprogramma. Un metodo alternativo per la gestione dell'overflow è di avere un registro di 1 bit asserito ogniqualvolta si verifica un overflow. Se un programmatore vuole controllare la presenza di un overflow deve includere un'istruzione esplicita di "salto se overflow asserito" dopo ogni istruzione aritmetica. Si tratta di una soluzione dispendiosa sia in tempo sia in spazio. Le trap fanno risparmiare tempo di esecuzione e memoria rispetto alla verifica affidata al controllo esplicito del programmatore.

Le trap potrebbero essere implementate tramite un test esplicito effettuato dal microprogramma (o dall'hardware); se viene rilevato un overflow, l'indirizzo di trap è caricato nel program counter. Ciò che fa scattare una trap a un certo livello potrebbe essere tenuto sotto controllo da un programma a un livello più basso. Il test effettuato dal microprogramma è ancora conveniente in termini di tempo rispetto al test svolto dal programmatore, perché può essere sovrapposto a qualche altra operazione. Inoltre consente un risparmio di memoria, perché basta che sia effettuato una sola volta, per esempio nel ciclo principale del microprogramma, indipendetemente dal numero d'istruzioni aritmetiche che ci sono nel programma principale.

Alcune delle condizioni che causano comunemente trap sono gli overflow e gli underflow (interi o in virgola mobile), le violazioni di protezione, gli opcode non definiti, gli overflow di stack, i tentativi di utilizzare dispositivi di I/O inesistenti, i tentativi di fetch di una parola da un indirizzo dispari, la divisione per zero.

5.6.5 Interrupt

Gli **interrupt** ("interruzioni") sono cambiamenti nel flusso esecutivo causati non dal programma in esecuzione, ma da qualche altro problema, in genere determinato dall'I/O. Per esempio un programma potrebbe ordinare al disco di iniziare il trasferimento delle informazioni e di inviare un interrupt non appena il trasferimento termini. Come le trap, gli interrupt interrompono il programma in esecuzione e trasferiscono il controllo a un gestore deputato a svolgere le azioni appropriate. Al loro compimento, il gestore di interrupt restituisce il controllo al programma interrotto. È suo compito far riprendere il processo interrotto esattamente dallo stesso stato in cui si trovava al momento dell'interruzione, il che implica il ripristino di tutti i registri interni allo stato precedente all'interrupt.

La differenza essenziale tra le trap e gli interrupt è che le *trap* sono sincrone al programma e gli *interrupt* sono asincroni. Se un programma viene rieseguito un milione di volte con lo stesso input, le trap si verificheranno sempre nello stesso punto, mentre gli interrupt possono variare a seconda, per esempio, del momento in cui viene premuto il tasto d'invio al terminale. La ragione alla base della riproducibilità delle trap e all'irriproducibilità degli interrupt è che le trap sono causate direttamente dal programma, gli interrupt, invece, sono causati dal programma per lo più indirettamente.

Per comprenderne meglio il funzionamento consideriamo un tipico esempio di interrupt: un calcolatore deve scrivere sullo schermo una riga di caratteri. Per prima cosa il software di sistema raccoglie in un buffer tutti i caratteri da scrivere, inizializza una variabile globale `ptr` in modo che punti all'inizio del buffer e quindi assegna a una seconda variabile globale `count` il numero di caratteri da visualizzare. Successivamente verifica che il terminale sia pronto e, in caso affermativo, invia il primo carattere (per mezzo di registri come quelli della Figura 5.31). Dopo aver fatto partire l'attività di I/O, la CPU è libera di eseguire un altro programma o di svolgere lavoro di altro genere.

Nel volgere di poco tempo, il carattere viene visualizzato e l'interrupt può essere lanciato. In forma semplificata, i passi si succedono nell'ordine seguente.

AZIONI HARDWARE

1. Il controllore del dispositivo attiva una linea di interrupt sul bus di sistema per dar via alla sequenza di interrupt.
2. Non appena pronta a gestire l'interrupt, la CPU attiva sul bus un segnale di conferma (acknowledgment) dell'interrupt.
3. Quando il controllore del dispositivo vede confermata la ricezione del proprio segnale di interrupt, invia sulla linea dati un piccolo intero che lo identifica. Questo numero si chiama **vettore di interrupt**.
4. La CPU preleva il vettore di interrupt dal bus e lo salva temporaneamente.
5. La CPU impila il program counter e il registro `PSW` sullo stack.

6. Quindi la CPU usa il vettore di interrupt come indice per individuare il nuovo program counter all'interno di una tabella posta all'inizio della memoria. Per esempio, se il program counter è di 4 byte, il vettore di interrupt *n* corrisponde all'indirizzo 4 *n*. Questo nuovo program counter punta all'inizio della routine di servizio dell'interrupt per il controllore che l'ha lanciato. Spesso anche `PSW` viene caricato o modificato (per esempio per disabilitare ulteriori interrupt).

AZIONI SOFTWARE

7. La routine di servizio dell'interrupt comincia con il salvare (sullo stack o in una tabella di sistema) tutti i registri che utilizza per poterli ripristinare successivamente.
8. In genere ogni vettore di interrupt è condiviso da tutti i dispositivi dello stesso tipo, perciò non identifica univocamente il terminale che ha causato l'interrupt. Si può risalire al numero del terminale attraverso la lettura di alcuni registri di dispositivo.
9. A questo punto è possibile leggere ogni altra informazione sull'interrupt, come il codice di stato.
10. Nel caso si fosse verificato un errore di I/O, lo si può gestire da questo momento.
11. Le variabili globali `ptr` e `count` vengono aggiornate; la prima è incrementata perché punti al byte successivo, la seconda è decrementata a indicare che manca un byte in meno da visualizzare. Se `count` è ancora maggiore di 0, ci sono altri caratteri da visualizzare, e quello ora puntato da `ptr` viene copiato nel registro buffer di output.
12. Se richiesto, viene inviato un codice speciale per specificare al dispositivo o al controllore dell'interrupt che l'interrupt è stato trattato.
13. Ripristino di tutti i registri salvati.
14. Esecuzione dell'istruzione `RITORNO DA INTERRUPT` che ripristina la modalità e lo stato della CPU a prima del sollevamento dell'interrupt. Infine il computer riprende la sua attività dal punto in cui era stato interrotto.

Un concetto chiave legato agli interrupt è la **trasparenza**. Al verificarsi di un interrupt si intraprendono alcune azioni e si procede con l'esecuzione di un certo codice, ma quando tutto è finito il computer deve ripartire esattamente dallo stato che aveva prima dell'interrupt. Una routine di interrupt che manifesta questa proprietà si dice *trasparente*. Avere interrupt trasparenti ne semplifica la comprensione.

Se un computer ha un solo dispositivo di I/O, gli interrupt funzionano sempre nel modo appena descritto, e non resta da aggiungere nulla a quanto detto. D'altro canto un grosso calcolatore può disporre di numerosi dispositivi di I/O, molti dei quali attivi nello stesso momento e magari per conto di utenti diversi. Esiste una probabilità non nulla che un secondo dispositivo di I/O voglia generare il *proprio* interrupt mentre una routine di interrupt è già in esecuzione.

In questa evenienza si possono prevedere due strategie. La prima è far sì che ogni routine di interrupt per prima cosa disabiliti eventuali interrupt successivi, ancor prima di salvare i registri. Questo approccio mantiene lo schema semplice, visto che gli interrupt sono trattati in modo strettamente sequenziale, ma può generare problemi ai dispositivi che non tollerano attese oltre una certa durata. Per esempio, su una linea di comunicazione a 9600 bps arrivano caratteri ogni 1042 μs, pronti o meno a riceverli. Se il primo carattere non è stato ancora elaborato, quando arriva il secondo è possibile che alcuni dati vadano persi.

Se un calcolatore possiede dispositivi di I/O per i quali la gestione del tempo è critica, una migliore strategia consiste nell'assegnare una priorità a ogni dispositivo di I/O: alta per i dispositivi molto critici, bassa per quelli meno critici. Allo stesso modo anche la CPU dovrebbe definire alcune priorità, in genere determinate da un campo in `PSW`. Quando un dispositivo di priorità *n* causa un'interruzione, anche la routine di interrupt dovrebbe girare a livello di priorità *n*.

Durante l'esecuzione di una routine di interrupt di priorità *n*, ogni tentativo di interruzione di dispositivi di priorità inferiore viene ignorato fintanto che la routine non sia stata completata e che la CPU non sia tornata a eseguire codice di priorità più bassa. Viceversa gli interrupt provenienti da dispositivi a priorità maggiore dovrebbero essere trattati senza attesa.

Essendo le stesse routine di interrupt soggette a interruzioni, il modo migliore per mantenerne una gestione corretta è far sì che tutti gli interrupt siano trasparenti. Si consideri un semplice esempio di interrupt multipli. Un computer ha tre dispositivi di I/O, una stampante, un disco e una linea (seriale) RS232, rispettivamente con priorità 2, 4 e 5. Al tempo $t = 0$ è in esecuzione un programma dell'utente, ma all'improvviso, al tempo $t = 10$, si verifica un interrupt dalla stampante. Viene allora fatta partire la routine di servizio dell'interrupt (*Interrupt Service Routine*, ISR) della stampante, come mostrato nella Figura 5.46.

Figura 5.46 Sequenza di interrupt.

Al tempo $t = 15$, la linea RS232 invoca attenzione e genera un interrupt. Dal momento che la linea RS232 ha priorità maggiore (5) della stampante (2), il suo interrupt si verifica. Lo stato della macchina, impegnata al momento nell'esecuzione della routine di servizio dell'interrupt della stampante, viene impilato sullo stack e parte l'esecuzione della routine di servizio dell'interrupt da RS232.

Poco dopo, all'istante $t = 20$, il disco richiede a sua volta di essere servito. Tuttavia la sua priorità (4) è minore di quella della routine di interrupt correntemente in esecuzione (5), perciò l'hardware della CPU non conferma l'interrupt, e questo resta sospeso in attesa. All'istante $t = 25$ la routine della linea RS232 termina e lo stato viene ripristinato al momento precedente l'interruzione da parte di RS232, ossia ritorna all'esecuzione della routine di servizio della stampante di priorità 2. Non appena la CPU passa alla priorità 2 e prima che ne venga eseguita una sola istruzione, il controllo passa all'interrupt del disco di priorità 4 e viene eseguita la sua routine di servizio. Al termine di essa, la routine della stampante ottiene di poter proseguire. Infine, al tempo $t = 40$, tutte le routine di servizio degli interrupt sono state completate e il programma dell'utente riprende da dove era stato interrotto.

A partire dall'8088, tutte le CPU Intel sono state dotate di due livelli (priorità) di interrupt: mascherabile e non mascherabile. Gli interrupt non mascherabili sono usati in genere solo per segnalare eventi quasi catastrofici, come gli errori di parità della memoria. Tutti i dispositivi di I/O usano interrupt mascherabili.

Quando un dispositivo di I/O lancia un interrupt, la CPU usa il vettore di interrupt per indicizzare una tabella di 256 elementi al fine di trovare l'indirizzo della sua routine di servizio. Gli elementi della tabella sono descrittori di segmenti di 8 byte e la tabella può trovarsi ovunque in memoria. Un registro globale punta al suo inizio.

Essendoci un solo livello di interrupt utilizzabile, la CPU non ha modo di far sì che un dispositivo di priorità maggiore interrompa una routine di servizio di priorità media, e di garantire al contempo che ciò sia vietato a un dispositivo di priorità bassa. Per risolvere questo problema, le CPU Intel sono usate spesso in congiunzione con un controllore esterno di interrupt (per esempio un 8259A). Quando sopraggiunge il primo interrupt, diciamo di priorità n, la CPU viene interrotta. Se poi segue un interrupt di priorità maggiore, il controllore di interrupt causa una seconda interruzione. Se invece il secondo interrupt è di livello inferiore, viene trattenuto fino al completamento del primo. Affinché questo schema funzioni, il controllore di interrupt deve sapere quando termina la routine di servizio corrente, così la CPU deve inviare un comando quando il trattamento dell'interrupt corrente viene completato.

5.7 Un esempio: le torri di Hanoi

Dopo aver studiato l'ISA di tre macchine diverse possiamo ora trarre le somme presentando l'esempio di programma delle torri di Hanoi nel caso delle due macchine più complesse. Abbiamo già offerto una versione Java di questo esempio nella Figura 5.42. Nei paragrafi seguenti proponiamo due programmi in codice assemblativo per le torri di Hanoi.

Ricorreremo a un piccolo espediente: invece di fornire la traduzione della versione Java per il Pentium 4 e per l'UltraSPARC III, ne daremo una traduzione dal C per ovviare ad alcuni problemi dell'I/O di Java. La sola differenza è la sostituzione della chiamata Java a `println` con l'istruzione C standard

```
printf("Move a disk from %d to %d\n", i, j)
```


A questo proposito non è rilevante la sintassi delle stringhe di formattazione della `printf` (fondamentalmente una stringa è interpretata letteralmente a eccezione della sequenza %d, che specifica il formato di visualizzazione decimale per l'intero successivo). Ciò che qui conta è che la procedura viene chiamata con tre parametri: una stringa di formattazione e due interi.

La ragione dell'utilizzo della versione C per il Pentium 4 e per l'UltraSPARC III è che la libreria Java di I/O non è disponibile in forma nativa per queste macchine, mentre la libreria C lo è. La differenza è minima e coinvolge la sola istruzione di stampa sullo schermo.

5.7.1 Le torri di Hanoi nel linguaggio assemblativo di Pentium 4

La Figura 5.47 fornisce una possibile traduzione della versione C delle torri di Hanoi per il Pentium 4; buona parte del codice è di comprensione immediata. Il registro `EBP` è usato come puntatore al record d'attivazione. Le prime due parole sono direttive per il linker, così il primo vero parametro, n (o N in questo caso, non essendo MASM case sensitive[1]) si trova alla posizione `EBP + 8`, seguito da i e j alle posizioni `EBP + 12` e `EBP + 16`, rispettivamente. La variabile locale k si trova in `EBP + 20`.

La procedura comincia con l'attivare un nuovo record alla fine del precedente. Per far ciò copia `ESP` nel puntatore al record d'attivazione, `EBP`. Quindi confronta n a 1 e salta alla clausola `else` se $n > 1$. Il ramo `then` impila sullo stack tre valori: l'indirizzo della stringa di formattazione, i e j, chiama `printf`.

I parametri sono impilati in ordine inverso, come richiesto dai programmi C. Ciò si rende necessario perché il puntatore alla stringa di formattazione si trovi in cima allo stack. Poiché `printf` ha un numero variabile di parametri, se i parametri fossero stati impilati in ordine, `printf` non avrebbe avuto modo di stabilire quante locazioni scorrere lungo lo stack per trovare la stringa di formattazione.

Dopo la chiamata si somma 12 a `ESP` per rimuovere i parametri dallo stack. Ovviamente essi non vengono davvero cancellati dalla memoria, ma la modifica di `ESP` li rende inaccessibili alle normali operazioni sullo stack.

La clausola `else`, che comincia a `L1`, è di facile comprensione. Per prima cosa calcola $6 - i - j$ e memorizza il valore in k. Qualsiasi sia il valore di i e j, il terzo piolo è sempre $6 - i - j$. Memorizzandone il valore in k ci si risparmia di doverlo ricalcolare una seconda volta.

Successivamente la procedura richiama se stessa tre volte, con parametri ogni volta differenti. Dopo ogni chiamata lo stack viene ripulito.

Le procedure ricorsive spesso confondono chi vi si avvicina per la prima volta, ma risultano facili da capire quando osservate a questo livello. Tutto ciò che succede è l'aggiunta dei parametri in cima allo stack e la chiamata stessa di procedura.

5.7.2 Le torri di Hanoi nel linguaggio assemblativo di UltraSPARC III

Ora riproviamo la traduzione, questa volta per l'UltraSPARC III. Il codice è elencato nella Figura 5.48. Dal momento che il codice dell'UltraSPARC III è particolarmente illeggibile, anche a livello del codice assemblativo e pur dopo molta pratica, ci siamo presi la libertà di

[1] Un linguaggio si definisce *case sensitive* se distingue tra caratteri minuscoli e maiuscoli, ovvero se considera a e A due identificativi diversi (*N.d.T.*).

```
        .586                            ; compila per il Pentium (invece che per l'8088, ecc.)
.MODEL FLAT
PUBLIC_towers                           ; esporta 'towers'
EXTERN_printf:NEAR                      ; importa printf
.CODE
_towers:  PUSH EBP                      ; salva EBP (puntatore al record d'attivazione) e decrementa ESP
          MOV EBP, ESP                  ; imposta nuovo puntatore al record al di sopra di ESP
          CMP [EBP+8] ,1                ; if (n == 1)
          JNE L1                        ; salta se n è diverso da 1
          MOV EAX, [EBP+16]             ; printf(" ...", i, j);
          PUSH EAX                      ; si noti che i parametri i, j e la stringa
          MOV EAX, [EBP+12]             ; di formattazione sono impilati sullo stack
          PUSH EAX                      ; in ordine inverso. È la convenzione per le chiamate C
          PUSH OFFSET FLAT:format       ; è l'indirizzo della stringa formato
          CALL_printf                   ; chiama printf
          ADD ESP, 12                   ; rimuove i parametri dallo stack
          JMP Done                      ; abbiamo concluso
L1:       MOV EAX, 6                    ; comincia k = 6 – i – j
          SUB EAX, [EBP+12]             ; EAX = 6 – i
          SUB EAX, [EBP+16]             ; EAX = 6 – i – j
          MOV [EBP+20], EAX             ; k = EAX
          PUSH EAX                      ; comincia towers(n – 1, i, k)
          MOV EAX, [EBP+12]             ; EAX = i
          PUSH EAX                      ; push di i
          MOV EAX, [EBP+8]              ; EAX = n
          DEC EAX                       ; EAX = n – 1
          PUSH EAX                      ; push di n – 1
          CALL_towers                   ; chiama towers(n – 1, i, 6 – i – j)
          ADD ESP, 12                   ; rimuove i parametri dallo stack
          MOV EAX, [EBP+16]             ; comincia towers(1, i, j)
          PUSH EAX                      ; push di j
          MOV EAX, [EBP+12]             ; EAX = i
          PUSH EAX                      ; push di i
          PUSH 1                        ; push di 1
          CALL_towers                   ; chiama towers(1, i, j)
          ADD ESP, 12                   ; rimuove i parametri dallo stack
          MOV EAX, [EBP+12]             ; comincia towers(n – 1, 6 – i – j, j)
          PUSH EAX                      ; push di j
          MOV EAX, [EBP+20]             ; EAX = k
          PUSH EAX                      ; push di k
          MOV EAX, [EBP+8]              ; EAX = n
          DEC EAX                       ; EAX = n – 1
          PUSH EAX                      ; push di n – 1
          CALL_towers                   ; chiama towers(n – 1, 6 – i – j, j)
          ADD ESP, 12                   ; modifica il puntatore allo stack
Done:     LEAVE                         ; preparazione all'uscita
          RET 0                         ; ritorno al chiamante
.DATA
format    DB "Move disk from %d to %d\n"  ; stringa di formattazione
END
```

Figura 5.47 Le torri di Hanoi nel linguaggio del Pentium 4.

cominciare con il definire alcuni simboli per renderlo più chiaro. Affinché possa funzionare, il programma deve essere fatto passare prima del suo assemblaggio attraverso un programma che si chiama *cpp*, un preprocessore C. Inoltre, in questo caso abbiamo usato lettere minu-


```
#define N %i0                       /* N è il parametro 0 in ingresso */
#define I %i1                       /* I è il parametro 1 in ingresso */
#define J %i2                       /* J è il parametro 2 in ingresso */
#define K %l0                       /* K è la variabile locale 0 */
#define Param0 %o0                  /* Param0 è il parametro 0 in uscita */
#define Param1 %o1                  /* Param1 è il parametro 1 in uscita */
#define Param2 %o2                  /* Param2 è il parametro 2 in uscita */
#define Scratch %l1                 /* NB. cpp usa per i commenti la convenzione C */
        .proc 04
        .global towers

towers: save %sp, –112, %sp
        cmp N, 1                    ! if(n == 1)
        bne Else                    ! if(n != 1) goto Else

        sethi %hi(format), Param0   ! printf("Move a disk from %d to %d\n", i, j)
        or Param0, %lo(format), Param0  ! Param0 = indirizzo della stringa di formato
        mov I, Param1               ! Param1 = i
        call printf                 ! chiama printf PRIMA di passare il parametro 2 (j)
        mov J, Param2               ! usa l'intervallo di ritardo dopo la chiamata per passare il parametro 2
        b Done                      ! abbiamo finito
        nop                         ! riempie l'intervallo di delay

Else:   mov 6, K                    ! comincia k = 6 – i – j
        sub K, J, K                 ! k = 6 – j
        sub K, I, K                 ! k = 6 – i – j

        add N, –1, Scratch          ! comincia towers(n – 1, i, k)
        mov Scratch, Param0         ! variabile di lavoro = n – 1
        mov I, Param1               ! parametro 1 = i
        call towers                 ! chiama towers PRIMA di passare il parametro 2 (k)
        mov K, Param2               ! usa l'intervallo di ritardo dopo la chiamata per passare il parametro 2

        mov 1, Param0               ! comincia towers(1, i, j)
        mov I, Param1               ! parametro 1 = i
        call towers                 ! chiama towers PRIMA di passare il parametro 2 (j)
        mov J, Param2               ! parametro 2 = j

        mov Scratch, Param0         ! comincia towers(n – 1, k, j)
        mov K, Param1               ! parametro 1 = k
        call towers                 ! chiama towers PRIMA di passare il parametro 2 (j)
        mov J, Param2               ! parametro 2 = j

Done:   ret                         ! ritorno
        restore                     ! usa l'intervallo di ritardo dopo ret per ripristinare le finestre

format: .asciz "Move a disk from %d to %d\n"
```

Figura 5.48 Le torri di Hanoi nel linguaggio dell'UltraSPARC III.

scole, perché l'assemblatore dell'UltraSPARC III ne richiede l'uso (casomai il lettore volesse provare a immettere il programma in una macchina UltraSPARC).

Dal punto di vista algoritmico la versione per l'UltraSPARC III è identica a quella per il Pentium 4. Entrambe cominciano con l'esaminare n e saltano alla clausola `else` se $n > 1$. La difficoltà principale nella versione per l'UltraSPARC III è dovuta ad alcune proprietà dell'ISA.

Per cominciare, l'UltraSPARC III deve passare l'indirizzo della stringa di formattazione alla *printf*, ma la macchina non può limitarsi a copiare l'indirizzo nel registro contenente il parametro in uscita, perché non c'è modo, con una sola istruzione, di mettere una costante di 32 bit in un registro, ma ce ne vogliono due: una `sethi` e una `or`.

Bisogna poi osservare che non è necessario modificare lo stack dopo la chiamata, perché la finestra di registri è messa a posto dall'istruzione `restore` alla fine della procedura. La possibilità di mettere i parametri nei registri senza dover accedere alla memoria è un grande vantaggio in termini di prestazioni se la profondità delle chiamate non è eccessiva, ma in generale l'intero meccanismo delle finestre di registri non giustifica la complessità che introduce.

Si noti ora l'istruzione `NOP` che segue il salto a `Done`. Si tratta di un intervallo di ritardo e questa istruzione viene sempre eseguita, anche se segue un'istruzione di salto incondizionato. Il problema è che l'UltraSPARC III ha una pipeline profonda e, al momento in cui l'hardware realizza di disporre di un'istruzione di salto, ha ormai praticamente completato l'istruzione successiva. Benvenuti nel Paese delle Meraviglie della Programmazione RISC.

Questa caratteristica stravagante si estende anche al caso delle chiamate di procedura. Osservate che la prima chiamata di `towers` nel ramo `else` pone $n-1$ in %o0 e i in %o1, ma effettua la chiamata di procedura `towers` prima di inserire l'ultimo parametro. Sul Pentium 4 per prima cosa si passano i parametri, poi si effettua la chiamata. Qui prima si passano alcuni parametri, quindi si fa la chiamata e infine si passa l'ultimo parametro. Anche in questo caso, quando la macchina si rende conto che sta trattando un'istruzione `call`, l'istruzione successiva è arrivata così in profondità nella pipeline che deve essere eseguita. Se così deve essere, perché non usare l'intervallo di ritardo per passare l'ultimo parametro? Anche se verrà usato dalla prima istruzione della procedura chiamata, il parametro sarà disponibile per tempo.

Infine, all'etichetta `Done` vediamo che anche l'istruzione `ret` ha un intervallo di ritardo, usato questa volta per l'istruzione `restore`, che incrementa `CWP` al fine di ripristinare la finestra di registri allo stato in cui se l'aspetta il chiamante.

5.8 Architettura IA-64 e Itanium 2

Intel sta raggiungendo velocemente il momento in cui avrà spremuto anche l'ultima goccia dell'ISA IA-32 e della linea di processori Pentium 4. I nuovi modelli possono sicuramente beneficiare dei progressi nelle tecnologie produttive che garantiscono transistor più ridotti (e quindi velocità di clock maggiori). Tuttavia diventa sempre più difficile escogitare nuovi stratagemmi per velocizzare ulteriormente le implementazioni, dato che i vincoli imposti dall'ISA IA-32 si rivelano sempre più costrittivi.

L'unica soluzione realistica è abbandonare IA-32 come linea principale di sviluppo e spostarsi verso un ISA completamente nuovo. È proprio ciò che Intel ha intenzione di fare: ci sono infatti progetti per due nuove linee. EMT-64 è un rifacimento ampliato del Pentium 4, con registri di 64 bit e spazio degli indirizzi a 64 bit. Questo processore risolve il problema dello spazio degli indirizzi, ma presenta ancora le complicazioni implementative del Pentium 4: in pratica può essere visto come un Pentium esteso.

L'altra nuova architettura, sviluppata congiuntamente da Intel e Hewlett Packard, si chiama **IA-64**. Si tratta di una macchina completamente a 64 bit, non di un'estensione di una macchina a 32 bit. Inoltre si differenzia radicalmente dal Pentium 4 per molti aspetti. Inizialmente si rivolge al mercato degli utenti di fascia alta, ma potrebbe alla lunga raggiungere il mondo dei desktop. A ogni modo, la sua architettura è così diversa da tutto quanto fin qui studiato che merita una trattazione a parte. La prima implementazione dell'architettura IA-64 è la serie Itanium. Nel resto del paragrafo analizzeremo l'architettura IA-64 e la CPU Itanium 2 che l'implementa.

5.8.1 Il problema di Pentium 4

Prima di addentrarci nei dettagli di IA-64 e dell'Itanium 2 è utile ricordare gli aspetti negativi del Pentium 4 per capire quali sono i problemi che Intel ha inteso risolvere con la nuova architettura. La questione fondamentale è che IA-32 è un ISA datato, le cui proprietà sono inadatte alla tecnologia corrente. È un ISA CISC con istruzioni di lunghezza variabile e una miriade di formati differenti difficili da decodificare velocemente. La tecnologia attuale lavora meglio con ISA RISC che hanno una sola lunghezza per le istruzioni e opcode di lunghezza fissa facili da decodificare. Le istruzioni IA-32 possono essere suddivise in microistruzioni di tipo RISC in fase di esecuzione, ma ciò richiede hardware (cioè superficie del chip), porta via tempo e contribuisce alla complessità del progetto. Questi sono i primi punti a sfavore.

Inoltre IA-32 è un ISA orientato alla memoria ed è a due indirizzi. La maggior parte delle istruzioni referenzia la memoria e la maggior parte dei programmatori e dei compilatori non si cura di far riferimento continuo alla memoria. La tecnologia attuale favorisce gli ISA load/store che accedono alla memoria solo per recuperare gli operandi e trasferirli nei registri, altrimenti effettuano tutta la computazione usando istruzioni a tre indirizzi di registri. Per di più questa deficienza peggiorerà con il passare del tempo, visto che la frequenza di clock delle CPU cresce molto più velocemente rispetto alla velocità della memoria. Questo è il secondo punto a sfavore.

Inoltre IA-32 dispone di un insieme di registri piccolo e irregolare. Non solo questo imbriglia i compilatori, ma il numero esiguo di registri d'uso generale (quattro o sei, a seconda che si contino anche `EDI` ed `ESI`) esige che i risultati intermedi siano riversati continuamente in memoria, risultando in un sovrannumero di accessi anche laddove non sarebbero necessari. E con questo terzo punto a sfavore IA-32 perde il primo tempo della partita.

Cominciamo ora il secondo tempo. Il numero limitato di registri causa molte dipendenze, in special modo dipendenze WAR non necessarie, perché i risultati devono essere messi da qualche parte e non c'è più spazio nei registri. Per aggirare la mancanza di registri bisogna che l'implementazione gestisca la rinomina internamente, un artificio tra i peggiori, per nascondere i registri all'interno del buffer di riordinamento. Per evitare di bloccarsi frequentemente a seguito di miss della cache, le istruzioni devono essere eseguite fuori sequenza. Tuttavia la semantica di IA-32 specifica precise interruzioni di modo tale che le istruzioni fuori sequenza siano ritirate in ordine. Tutto ciò richiede però altro hardware molto complesso. Quarto punto a sfavore.

Per svolgere velocemente tutte queste operazioni occorre una pipeline con molti stadi. A sua volta, l'utilizzo di queste pipeline comporta che le istruzioni impieghino molti cicli per venir completate. Di conseguenza per garantire che le istruzioni introdotte nelle pipeline

siano quelle giuste è essenziale una predizione dei salti molto precisa. Una predizione sbagliata richiede lo svuotamento della pipeline, un'operazione molto costosa, perciò anche un tasso di predizioni erronee abbastanza basso può causare un degrado sostanziale delle prestazioni. Quinto punto a sfavore.

Per alleviare il problema delle predizioni sbagliate il processore deve effettuare l'esecuzione speculativa, con tutti i problemi che comporta, specie quando i riferimenti alla memoria causano un'eccezione. Sesto punto a sfavore.

Non occorre continuare per capire che ci troviamo davanti a un vero problema. Non abbiamo neanche menzionato il fatto che gli indirizzi di 32 bit di IA-32 limitano i singoli programmi a 4 GB di memoria, un limite problematico su server di fascia alta. EMT-64 risolve questo problema, ma non tutti gli altri.

Dopo tutto la situazione di IA-32 può essere paragonata con ragione allo stato della meccanica celeste appena prima di Copernico. La teoria astronomica allora dominante stabiliva che la Terra fosse immobile e che i pianeti ruotassero con epicicli attorno a essa. D'altra parte, al migliorare delle osservazioni e al crescere delle discrepanze tra i dati osservati e il modello, furono aggiunti epicicli a epicicli finché l'intero modello collassò a causa della sua complessità interna.

Intel si trova ora nello stesso guaio. Una porzione enorme dei transistor del Pentium 4 è destinata a scomporre le istruzioni CISC, stabilire ciò che può essere svolto in parallelo, risolvere conflitti, fare predizioni, rimediare alle conseguenze di predizioni scorrette e ad altra ordinaria amministrazione, lasciando una frazione sorprendentemente piccola per svolgere i compiti che l'utente ha effettivamente richiesto. La conclusione cui è giunta Intel è l'unica sensata: buttare via tutto (IA-32) e ricominciare da capo dopo aver fatto tabula rasa (IA-64). EMT-64 consente un certo margine di respiro, ma si tratta davvero di nascondere il problema della complessità sotto un tappeto di novità.

5.8.2 Modello IA-64 e calcolo che utilizza il parallelismo esplicito

L'idea chiave alla base di IA-64 è di spostare il carico di lavoro dalla fase di esecuzione alla fase di compilazione. Nel Pentium 4, durante l'esecuzione la CPU riordina le istruzioni, rinomina i registri, fa lo scheduling delle unità funzionali e svolge molto altro lavoro per stabilire come occupare pienamente tutte le risorse hardware. Nel modello IA-64 il compilatore prevede tutto ciò e produce un programma che può essere eseguito così com'è, senza che l'hardware debba destreggiarsi tra tutti quei dettagli durante l'esecuzione. Per esempio, invece di dire al compilatore che la macchina ha otto registri quando in realtà ne ha 128, per poi cercare di evitare le dipendenze durante l'esecuzione, il modello IA-64 comunica al compilatore il numero effettivo di registri della macchina, così da poter produrre programmi senza conflitti tra registri. Analogamente, in questo modello il compilatore tiene traccia delle unità funzionali occupate e non emette istruzioni che usano unità funzionali non disponibili. Il modello che rende il parallelismo sottostante nell'hardware visibile al compilatore si chiama **EPIC** (*Explicitly Parallel Instruction Computing*, "calcolo che utilizza il parallelismo esplicito"). EPIC può essere considerato in un certo senso il successore di RISC.

Il modello IA-64 presenta numerose funzionalità per incrementare le prestazioni. Tra queste ce ne sono alcune per ridurre i riferimenti in memoria, per lo scheduling delle istruzioni, per ridurre i salti condizionati e altro. Vediamole ora una per una, riferendo come sono implementate nell'Itanium 2.

5.8.3 Riduzione degli accessi in memoria

L'Itanium 2 ha un modello di memoria semplice. La memoria consiste di 2^{64} byte di memoria lineare. Ci sono istruzioni per accedere a unità di memoria di 1, 2, 4, 16 e 10 byte, quest'ultima per i numeri in virgola mobile IEEE 745 di 80 bit. I riferimenti in memoria non devono essere allineati alle loro estremità naturali, ma se non lo sono ciò comporta un degrado delle prestazioni. L'ordinamento della memoria può essere sia little-endian sia big-endian, specificato tramite il valore di un bit appartenente a un registro a disposizione del sistema operativo.

Nei computer moderni l'accesso alla memoria costituisce un collo di bottiglia molto stretto, perché le CPU sono molto più veloci della memoria. Un modo per ridurre gli accessi alla memoria è disporre di una grande cache di primo livello sul chip e di una cache di secondo livello ancora più grande, vicina al chip. Oltre all'uso delle cache ci sono altri modi per ridurre gli accessi in memoria, alcuni dei quali sono sfruttati da IA-64.

Il modo migliore per velocizzare i riferimenti in memoria è... non usarli per niente. L'implementazione Itanium 2 del modello IA-64 ha 128 registri d'uso generale, di 64 bit ciascuno. I primi 32 registri sono statici, mentre i rimanenti 96 sono usati come registri di stack, in un modo molto simile allo schema a finestra di registri dell'UltraSPARC III. A differenza dell'UltraSPARC il numero di registri visibili al programma è però variabile e può cambiare da procedura a procedura. Così ogni procedura ha accesso a 32 registri statici più un numero (variabile) di registri allocati dinamicamente.

All'atto della chiamata di una procedura il puntatore allo stack è incrementato, affinché i parametri d'ingresso siano visibili nei registri, ma non viene allocato alcun registro per le variabili locali. È la procedura stessa a decidere quanti registri necessita, e per allocarli incrementa lo stack pointer. Questi registri non devono essere salvati all'ingresso né ripristinati all'uscita, anche se la procedura deve aver cura di salvare e poi ripristinare i registri statici da modificare. Grazie al numero variabile di registri disponibili, legato alle reali esigenze di ogni procedura, si evita lo spreco dei già pochi registri; le chiamate di procedura possono così innestarsi a maggior profondità prima che si debbano riversare i registri in memoria.

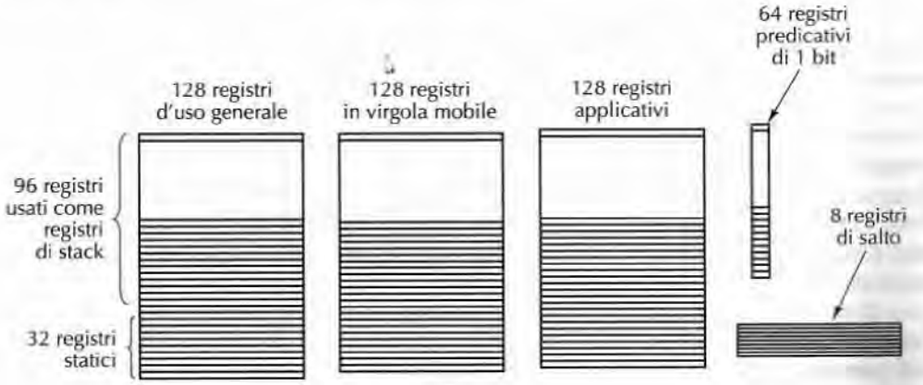


Figura 5.49 Registri di Itanium 2.

L'Itanium 2 dispone anche di 128 registri in virgola mobile che rispettano il formato IEEE 745. Non vengono usati come registri di stack, ma il loro numero abbondante consente di memorizzare i risultati intermedi di molte operazioni in virgola mobile, senza bisogno di salvarli temporaneamente in memoria.

Ci sono anche 64 registri predicativi di 1 bit, otto registri di salto e 128 registri per applicazioni specifiche usati per vari propositi, come il passaggio di parametri tra programmi applicativi e il sistema operativo. La Figura 5.49 dà una panoramica dei registri dell'Itanium 2.

5.8.4 Scheduling delle istruzioni

Uno dei problemi principali del Pentium 4 è la difficoltà insita nel trovare uno scheduling delle varie istruzioni, a favore delle diverse unità funzionali, che eviti le dipendenze. Ci vogliono meccanismi esageratamente complicati per gestire tutti questi aspetti in fase d'esecuzione, e così un'ingente porzione dell'area del chip è dedicata al loro trattamento. IA-64 e Itanium 2 evitano questi problemi scaricando il lavoro sul compilatore. L'idea chiave è che un programma consista in una sequenza di **gruppi d'istruzioni.** Entro un certo limite le istruzioni di un gruppo non sono mai in conflitto tra di loro, non usano più unità funzionali e risorse di quelle a disposizione della macchina, non contengono dipendenze RAW e WAW, ma solo dipendenze WAR limitate. I gruppi consecutivi d'istruzioni danno l'impressione di un'esecuzione strettamente sequenziale, laddove l'esecuzione del secondo gruppo non comincia finché non sia stato completato il primo. Tuttavia la CPU può iniziare a elaborare (in parte) il secondo gruppo non appena lo reputi sicuro.

Una conseguenza di queste regole è che la CPU è libera di scegliere l'ordine di esecuzione delle istruzioni (inclusa l'esecuzione parallela), senza timore di conflitti. Il comportamento di un programma con un gruppo d'istruzioni che viola le regole non è definito. Spetta al compilatore riordinare il codice assemblativo generato dal programma sorgente al fine di soddisfare tutti i requisiti. In fase di sviluppo dell'applicazione il compilatore può mettere ciascuna istruzione in un gruppo per ottenere una compilazione rapida; è una soluzione facile, ma che porta a prestazioni molto scarse. Al momento di produrre il codice definitivo il compilatore può impiegare molto tempo per ottimizzarlo.

Le istruzioni sono organizzate in **pacchetti d'istruzioni** (*bundle*) di 128 bit, come mostrato nella parte alta della Figura 5.50. Ogni pacchetto contiene tre istruzioni di 41 bit e un campo template di 5 bit. Un gruppo d'istruzioni non deve prendere necessariamente un numero intero di pacchetti; può cominciare e terminare nel mezzo di un pacchetto.

Esistono più di cento formati d'istruzioni. La Figura 5.50 ne mostra uno tipico per le operazioni della ALU, come la `ADD` che somma due registri in un terzo. Il primo campo `GRUPPO DI OPERAZIONI` è il gruppo principale e specifica in sostanza la classe generale di appartenenza dell'istruzione, per esempio le operazioni ALU intere. Il campo successivo, `TIPO DI OPERAZIONE`, restituisce la particolare operazione richiesta, per esempio `ADD` o `SUB`. Quindi vengono i tre campi registro e infine c'è il `REGISTRO PREDICATIVO`, trattato a breve.

Il template del pacchetto specifica le unità funzionali necessarie al pacchetto e anche l'eventuale presenza e posizione di un'estremità del pacchetto d'istruzioni. Le unità funzionali principali sono le ALU (intera e non), le operazioni di memoria, le operazioni floating point e i salti. Ovviamente l'ortogonalità completa con sei unità e tre istruzioni richiederebbe 216 combinazioni, più altre 216 per indicare un marcatore di gruppo d'istruzioni dopo

Figura 5.50 Pacchetto d'istruzioni IA-64 contenente tre istruzioni.

l'istruzione 0, altre 216 per indicare un marcatore di gruppo d'istruzioni dopo l'istruzione 1, e ancora 216 per indicare un marcatore di gruppo d'istruzioni dopo l'istruzione 2. Con soli 5 bit disponibili è consentito solo un numero molto limitato di queste combinazioni. C'è da dire che, se anche ci fosse un modo di specificare tre istruzioni in virgola mobile in un pacchetto d'istruzioni, non servirebbe a nulla, perché la CPU non può avviare simultaneamente tre istruzioni in virgola mobile. Le combinazioni previste sono quelle effettivamente attuabili.

5.8.5 Riduzione dei salti condizionati: attribuzione di predicati

Un'altra caratteristica importante di IA-64 è il modo nuovo di trattare i salti condizionati. Se ci fosse un modo di liberarsene quasi del tutto, le CPU ne beneficierebbero grandemente in semplicità e velocità. Di primo acchito si direbbe impossibile sbarazzarsi dei salti condizionati, perché i programmi sono pieni d'istruzioni `if`, e invece IA-64 usa una tecnica, detta **attribuzione di predicati** (*predication*), che può ridurre di molto il loro numero (August et al., 1998; Hwu, 1998). Descriviamola brevemente.

Nei processori attuali tutte le istruzioni sono incondizionate, nel senso che quando la CPU raggiunge un'istruzione non fa altro che svolgerla. Non c'è alcun dubbio del tipo "fare o non fare, questo è il dilemma". Al contrario, in un'architettura predicativa le istruzioni contengono condizioni (predicati) che stabiliscono se devono essere eseguite o meno. Questo cambio di paradigma da istruzioni incondizionate a istruzioni predicative rende possibile la riduzione di (molti) salti condizionati. Invece di dover scegliere tra due sequenze d'istruzioni incondizionate, si fondono tutte le istruzioni in una sola sequenza d'istruzioni predicative, usando predicati diversi per istruzioni diverse.

Vediamo come funziona l'attribuzione di predicati a partire dalla Figura 5.51, che mostra un semplice esempio di **esecuzione condizionata**, concetto precursore dell'attribuzione di predicati. La Figura 5.51(a) contiene un'istruzione `if`, mentre la Figura 5.51(b) ne mostra la traduzione in tre istruzioni: un confronto, un salto condizionato e un trasferimento.

Nella Figura 5.51(c) ci liberiamo del salto condizionato grazie a una nuova istruzione, `CMOVZ`, che è un trasferimento condizionato. Il suo funzionamento è controllare se il terzo registro, `R1`, vale 0; se è così copia `R3` in `R2`, altrimenti non esegue nulla.

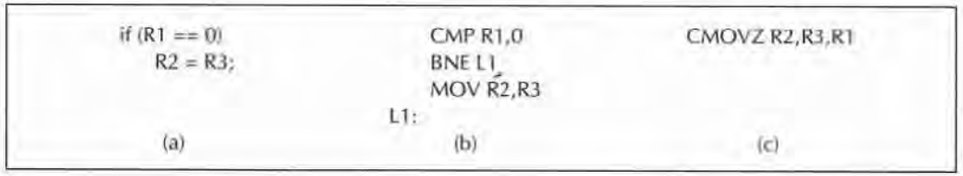


Figura 5.51 (a) Un'istruzione if. (b) Codice assemblativo per (a). (c) Un'istruzione condizionata.

Una volta resa disponibile un'istruzione che può copiare dati quando un certo registro vale 0, ci vuole poco per avere un'istruzione che trasferisce dati quando un certo registro non è 0: chiamiamola `CMOVN`. Con queste due istruzioni siamo sulla buona strada per l'esecuzione pienamente condizionata. Si immagini un'istruzione `if` con molti assegnamenti nel ramo `then` e molti altri nel ramo `else`. L'intera istruzione può essere tradotta in codice che imposta un certo registro a 0 se la condizione è falsa e a un altro valore se è vera. A seguire, il ramo `then` può essere compilato in una sequenza d'istruzioni `CMOVN` e il ramo `else` in una sequenza d'istruzioni `CMOVZ`.

Tutte queste istruzioni, l'impostazione del registro, le `CMOVN` e le `CMOVZ`, formano un singolo blocco elementare senza salti condizionati. L'ordine delle istruzioni può addirittura essere modificato, sia dal compilatore (che può anticipare gli assegnamenti prima del test) sia durante l'esecuzione. L'unico limite è che il valore della condizione deve essere noto prima che le istruzioni condizionate siano ritirate (verso la fine della pipeline). La Figura 5.52 illustra un semplice esempio con un ramo `then` e un ramo `else`.

Fin qui abbiamo presentato istruzioni condizionate molto semplici (appartenenti in realtà al Pentium 4), ma si tenga presente che tutte le istruzioni di IA-64 sono predicative. Ciò significa che l'esecuzione di ogni istruzione può essere resa condizionata. I 6 bit addizionali cui si è fatto riferimento prima servono a selezionare uno dei 64 registri predicativi di 1 bit. Perciò un'istruzione `if` sarà compilata in codice che imposta a 1 uno dei registri predicativi se la condizione è vera, e a 0 viceversa. Allo stesso tempo imposta automaticamente un altro registro predicativo al valore opposto. Grazie all'attribuzione di predicati le istruzioni macchina che costituiscono le clausole `then` ed `else` possono essere fuse in un solo flusso d'istruzioni, dove le prime si avvalgono del predicato originale, e le seconde del suo opposto.

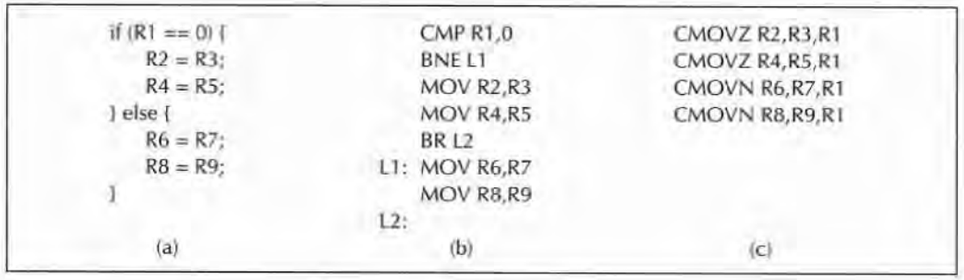


Figura 5.52 (a) Un'istruzione if. (b) Codice assemblativo per (a). (c) Esecuzione condizionata.

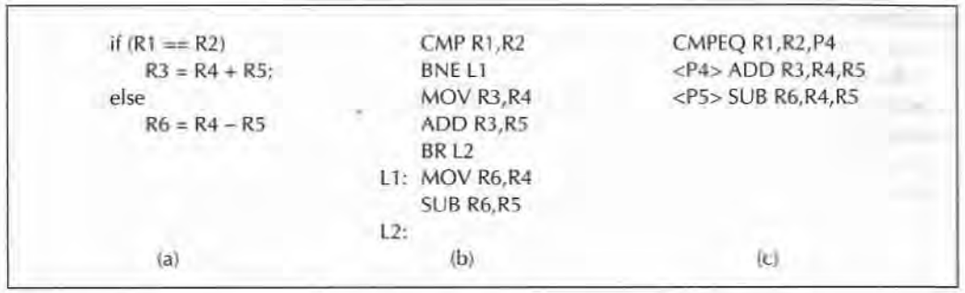


Figura 5.53 (a) Un'istruzione if. (b) Codice assemblativo per (a). (c) Esecuzione predicativa.

Per quanto semplice, l'esempio della Figura 5.53 mostra l'idea fondamentale di come l'attribuzione di predicati sia utilizzabile per l'eliminazione dei salti. L'istruzione `CMPEQ` confronta due registri e asserisce il registro predicativo `P4` se sono uguali, lo azzera se sono diversi. Inoltre imposta un registro accoppiato, diciamo `P5`, secondo la condizione complementare, dopo di che è possibile elencare di seguito le istruzioni delle parti `then` ed `else`, ciascuna condizionata da un certo registro predicativo (indicato tra parentesi angolari). Lo schema si può applicare a qualsiasi tipo d'istruzione, purché opportunamente predicata.

IA-64 conduce questa idea alle sue estreme conseguenze, disponendo d'istruzioni di confronto che impostano registri predicativi e d'istruzioni aritmetiche e non, la cui esecuzione dipende da un certo registro predicativo. Le istruzioni predicative possono essere inserite nella pipeline in sequenza, senza stalli né altri problemi, ragion per cui sono così utili.

L'attribuzione di predicati su IA-64 avviene facendo sì che tutte le istruzioni siano eseguite. In fondo alla pipeline, al momento di ritirare un'istruzione si verifica se il suo predicato è vero. In caso affermativo l'istruzione è ritirata normalmente e il suo risultato è salvato nel registro destinazione. Se invece il predicato è falso, non si effettua alcun write-back e l'istruzione non ha effetto. L'attribuzione di predicati è esaminata in maggior dettaglio in Dulong (1998).

5.8.6 Caricamenti speculativi

Un'altra caratteristica di IA-64 che ne velocizza l'esecuzione è la presenza di `LOAD` speculative. Se un'istruzione `LOAD` è speculativa e non va a buon fine, invece di causare un'eccezione, semplicemente si ferma e viene asserito un bit associato al registro da caricare che lo segnala come invalido. Si tratta proprio del poison bit introdotto nel Capitolo 4. Se il registro associato al poison bit viene usato successivamente, allora viene sollevata un'eccezione, che altrimenti non si verifica.

Il modo usuale di impiegare la speculazione è di permettere al compilatore di anticipare istruzioni `LOAD` in posizioni antecedenti al loro effettivo bisogno. Se cominciano prima è probabile che il loro risultato sia disponibile al momento opportuno. Il compilatore inserisce un'istruzione `CHECK` laddove necessita di un registro appena caricato. Se il valore è disponibile la `CHECK` si comporta come una `NOP` e l'esecuzione prosegue immediatamente. Se invece il valore non è ancora disponibile, l'istruzione successiva deve

andare in stallo. Se si è verificata un'eccezione e il poison bit è asserito, anche l'eccezione sospesa viene sollevata.

In conclusione, le macchine che implementano l'architettura IA-64 devono la loro velocità a diversi fattori. Al centro c'è il motore RISC a tre indirizzi, con pipeline, di tipo load/store e che è allo stato dell'arte della materia. In più IA-64 è provvista di un modello di parallelismo esplicito che richiede un compilatore in grado di stabilire quali istruzioni eseguire contemporaneamente senza conflitti e quindi capace di raggrupparle in pacchetti d'istruzioni. Così facendo la CPU può limitarsi alla pura gestione dello scheduling di un pacchetto senza inutili ripensamenti. Inoltre, l'attribuzione di predicati consente di fondere le istruzioni di entrambi i rami di un'istruzione `if` in un solo flusso, eliminando i salti condizionati, e quindi la necessità di predire quale ramo verrà percorso. Infine le `LOAD` speculative rendono possibile il fetch anticipato degli operandi, senza penalità in caso risultassero inutili.

Per maggiori informazioni sull'Itanium 2 e sulla sua microarchitettura si veda (McNairy e Soltis, 2003; Rusu et al., 2004).

5.9 Riepilogo

L'architettura dell'insieme d'istruzioni è ciò a cui molte persone pensano in temini di "linguaggio macchina". A questo livello la macchina ha una memoria orientata al byte o alla parola, costituita da decine di megabyte, e dispone d'istruzioni quali `MOVE`, `ADD` e `BEQ`.

Gran parte dei calcolatori moderni dispone di una memoria organizzata come una sequenza di byte, con gruppi di 4 o 8 byte a formare le parole. In genere ci sono tra gli 8 e i 32 registri, ciascuno lungo una parola. Su alcune macchine (come il Pentium 4) i riferimenti in memoria non devono essere allineati alle estremità naturali delle parole di memoria, mentre su altri (come l'UltraSPARC III) ciò è richiesto.

Le istruzioni hanno in genere uno, due o tre operandi, indirizzati in modo immediato, diretto, a registro, indicizzato o in altri modi. Alcune macchine dispongono di un gran numero di complesse modalità d'indirizzamento. Ci sono in genere istruzioni per trasferire dati, operazioni unarie e binarie, comprese le operazioni aritmetiche e quelle booleane, salti, chiamate di procedura, cicli e alle volte istruzioni per l'I/O. Le istruzioni in genere trasferiscono una parola dalla memoria in un registro (o viceversa), sommano, sottraggono, moltiplicano o dividono due registri o un registro e una parola di memoria, oppure confrontano due elementi nei registri o in memoria. Non è insolito trovare computer con più di 200 istruzioni nel loro repertorio. Le macchine CISC ne hanno spesso molte di più.

Il controllo del flusso a livello 2 si realizza tramite una varietà di primitive, inclusi i salti, le chiamate di procedura, le chiamate di coroutine, le trap e gli interrupt. I salti sono usati per concludere una sequenza d'istruzioni e cominciarne una nuova. Le procedure sono un'astrazione di questo meccanismo e consentono di isolare una parte di un programma come un'unità che può essere richiamata da molti punti diversi. Le coroutine permettono l'esecuzione simultanea di due thread di controllo. Le trap sono utilizzate per segnalare situazioni eccezionali, come l'overflow aritmetico. Gli interrupt permettono all'I/O di interagire concorrentemente con la computazione principale: la CPU riceve un segnale non appena l'I/O ha completato la sua attività.

Le torri di Hanoi costituiscono un problema semplice e divertente che ammette l'elegante soluzione ricorsiva che abbiamo esaminato.

Infine, l'architettura IA-64 usa il modello di calcolo EPIC per facilitare l'uso del parallelismo da parte dei programmi. Per migliorare le prestazioni si avvale di gruppi d'istruzioni, dell'attribuzione di predicati e di `LOAD` speculative. In definitiva potrebbe rappresentare un significativo miglioramento del Pentium 4, anche se scarica gran parte del peso della parallelizzazione sul compilatore.

Problemi

1. Una parola di un computer little-endian contiene il valore numerico 3. Se viene trasmessa a un computer big-endian byte per byte e ivi memorizzata, con il byte 0 al posto del byte 0 e così via, quale sarà il suo valore numerico nella macchina big-endian?
2. Le istruzioni del Pentium 4 possono contenere qualsiasi numero di byte, che sia pari o dispari. Nell'UltraSPARC III tutte le istruzioni contengono un numero intero di parole, cioè un numero pari di byte. Si trovi un punto a favore dello schema del Pentium 4.
3. Si progetti un opcode espandibile, in cui le seguenti istruzioni possano essere tutte codificate in istruzioni di 36 bit:

   7 istruzioni con due indirizzi di 15 bit e un identificativo di registro di 3 bit

   500 istruzioni con un indirizzo di 15 bit e un identificativo di registro di 3 bit

   40 istruzioni senza indirizzi né registri.
4. Una certa macchina dispone d'istruzioni di 16 bit e d'indirizzi di 6 bit. Alcune istruzioni contengono un indirizzo, altre due. Se ci sono n istruzioni a due indirizzi, qual è il numero massimo d'istruzioni a un indirizzo?
5. È possibile progettare un opcode espandibile che consenta di codificare le seguenti istruzioni in istruzioni di 12 bit? Un registro è identificato da 3 bit.

   4 istruzioni con tre registri

   255 istruzioni con un registro

   16 istruzioni senza registri
6. Dati i seguenti valori di memoria e una macchina a un indirizzo dotata di accumulatore, quali valori sono caricati all'interno dell'accumulatore dalle istruzioni specificate?

   la parola 20 contiene 40

   la parola 30 contiene 50

   la parola 40 contiene 60

   la parola 50 contiene 70

   a. `LOAD IMMEDIATE 20`
   b. `LOAD DIRECT 20`
   c. `LOAD INDIRECT 20`
   d. `LOAD IMMEDIATE 30`
   e. `LOAD DIRECT 30`
   f. `LOAD INDIRECT 30`

7. Si confrontino le macchine a 0, 1, 2 e 3 indirizzi scrivendo un programma per il calcolo di

$$X = (A + B \times C) / (D - E \times F)$$

per ciascuna delle quattro macchine. Le istruzioni disponibili sono:

0 Indirizzi	1 Indirizzo	2 Indirizzi	3 Indirizzi
`PUSH M`	`LOAD M`	`MOV` (X = Y)	`MOV` (X = Y)
`POP M`	`STORE M`	`ADD` (X = X + Y)	`ADD` (X = X + Y)
`ADD`	`ADD M`	`SUB` (X = X – Y)	`SUB` (X = X – Y)
`SUB`	`SUB M`	`MUL` (X = X * Y)	`MUL` (X = X * Y)
`MUL`	`MUL M`	`DIV` (X = X / Y)	`DIV` (X = X / Y)
`DIV`	`DIV M`		

M è un indirizzo di memoria di 16 bit, *X*, *Y* e *Z* sono o indirizzi di 16 bit o registri di 4 bit. La macchina a 0 indirizzi usa uno stack, la macchina a 1 indirizzo usa un accumulatore, mentre le altre due hanno 16 registri e istruzioni per operare su tutte le possibili combinazioni di locazioni di memoria e registri. `SUB X, Y` sottrae *Y* da *X* e `SUB X, Y, Z` sottrae *Z* da *Y* e pone il risultato in *X*. Con opcode di 8 bit e lunghezze d'istruzioni che sono multiple di 4 bit, di quanti bit ha bisogno ciascuna macchina per calcolare *X*?

8. Si escogiti un meccanismo d'indirizzamento che consenta di specificare, in uno spazio d'indirizzi molto grande, un insieme arbitrario di 64 indirizzi non necessariamente contigui, mediante un campo di 6 bit.

9. Si indichi uno svantaggio del codice auto-modificante non menzionato nel testo.

10. Si convertano le formule seguenti da notazione infissa a polacca inversa.

a. A + B + C + D – E

b. (A – B) × (C + D) + E

c. (A × B) + (C × D) + E

d. (A – B) × (((C – D × E) / F) / G) × H

11. Quali delle seguenti coppie di formule in notazione polacca inversa sono matematicamente equivalenti?

a. A B + C + e A B C + +

b. A B – C – e A B C – –

c. A B × C + e A B C + ×

12. Si convertano le seguenti formule da notazione polacca inversa a notazione infissa.

a. A B – C + D ×

b. A B / C D / +

c. A B C D E + × × /

d. A B C D E × F / + G – H / × +

13. Si scrivano tre formule in notazione polacca inversa che non possono essere convertire in notazione infissa.

14. Si convertano le seguenti formule booleane infisse in notazione polacca inversa.

a. (A AND B) OR C

b. (A OR B) AND (A OR C)

c. (A AND B) OR (C AND D)

15. Si converta la seguente formula infissa in notazione polacca inversa e si scriva del codice IJVM per valutarla.

$$(5 \times 2 + 7) – (4 / 2 + 1)$$

16. L'istruzione in linguaggio assemblativo

```
MOV REG,ADDR
```

del Pentium 4 carica un registro dalla memoria. Invece l'istruzione dell'UltraSPARC III che carica un registro dalla memoria è

```
LOAD ADDR,REG
```

Perché l'ordine degli operandi è diverso?

17. Quanti registri ci sono nella macchina che ha i formati d'istruzione della Figura 5.25?

18. Nella Figura 5.25 il bit 23 è usato per distinguere il formato 1 dal formato 2. Non c'è alcun bit per distinguere il formato 3. Come fa l'hardware a sapere quando usarlo?

19. Capita spesso nella programmazione di dover determinare dove si posiziona una variabile *X* rispetto a un intervallo [*A*, *B*]. Se si dispone di un'istruzione a tre indirizzi con operandi *A*, *B* e *X*, quanti bit codici di condizione si dovranno impostare per rappresentare il risultato del confronto?

20. Il Pentium 4 ha un bit codice di condizione che tiene traccia del riporto oltre il terzo bit a seguito di un'operazione aritmetica. A che cosa serve?

21. L'UltraSPARC III non dispone di un'istruzione per caricare un numero di 32 bit in un registro. Per farlo si avvale in genere di una sequenza di due istruzioni, `SETHI` e `ADD`. C'è un unico modo per caricare un numero di 32 bit in un registro o ce ne sono degli altri? Si motivi la risposta.

22. Un tuo amico ti ha appena svegliato telefonandoti nel cuore della notte e ti ha messo a parte delle sua nuova idea brillante: un'istruzione con due opcode. Gli consigli di correre all'ufficio brevetti o di tornare sui banchi di scuola?

23. L'8051 non dispone d'istruzioni con offset maggiori di 8 bit. Questo implica che non può indirizzare locazioni di memoria oltre la 255? E se può, come lo fa?

24. Confronti del tipo

```
if (k == 0) ...
if (a > b) ...
if (k < 5) ...
```

sono comuni in programmazione. Si progetti un'istruzione che effettui questi test in modo efficiente: quali sono i campi dell'istruzione?

25. Dato il numero binario di 16 bit 1001 0101 1100 0011, si mostrino gli effetti delle seguenti operazioni:

a. uno scorrimento verso destra di 4 bit con riempimento di zeri

b. uno scorrimento verso destra di 4 bit con estensione del segno

c. uno scorrimento verso sinistra di 4 bit

d. una rotazione verso sinistra di 4 bit

e. una rotazione verso destra di 4 bit.

26. Come si può azzerare una parola di memoria su una macchina priva d'istruzione `CLR`?

27. Si calcoli il valore dell'espressione booleana (*A* AND *B*) OR *C* per

A = 1101 0000 1010 0011
B = 1111 1111 0000 1111
C = 0000 0000 0010 0000

28. Si escogiti un modo per scambiare il valore di due variabili *A* e *B* senza usare una terza variabile o registro. *Suggerimento*: si pensi all'operazione di OR ESCLUSIVO.

29. Su un certo computer è possibile svolgere le seguenti operazioni in meno tempo di quanto impiegato da una moltiplicazione: trasferire un numero da un registro all'altro, far scorrere entrambi di un certo quantitativo di bit e sommare i risultati. Sotto quali condizioni questa sequenza d'istruzioni è utile per calcolare "costante × variabile"?

30. Macchine diverse hanno diverse densità d'istruzioni (numero di byte richiesti per svolgere un certo calcolo). Tradurre i seguenti frammenti di codice Java nei linguaggi assemblativi del Pentium 4, dell'UltraSPARC III e in linguaggio IJVM. Calcolare quindi il numero di byte richiesti da ciascuna macchina per calcolare le espressioni. Si ipotizzi che *i* e *j* sono variabili locali in memoria, per il resto si facciano sempre le assunzioni più ottimistiche

a. `i = 3;`

b. `i = j;`

c. `i = j - 1;`

31. Le istruzioni di ciclo illustrate nel testo servono per la gestioni di cicli `for`. Progettare invece un'istruzione che potrebbe essere utile per gestire comuni cicli `while`.

32. Se i monaci di Hanoi possono spostare un disco al minuto (non hanno fretta di portare a termine il compito perché le loro professionalità sono poco apprezzate sul mercato del lavoro di Hanoi), quanto impiegheranno a risolvere il problema con 64 dischi? Esprimere il risultato in anni.

33. Per quale motivo i dispositivi di I/O inviano il vettore di interrupt sul bus? Non potrebbero invece memorizzare questa informazione in una tabella di memoria?

34. Un certo computer usa il meccanismo DMA per leggere dal disco. Il disco ha 64 settori di 512 byte per ogni traccia. Il tempo di rotazione del disco è 16 ms. Il bus è largo 16 bit e ogni trasferimento sul bus impiega 500 ns. Un'istruzione di CPU richiede in media due cicli di bus. In che misura la CPU viene rallentata dal DMA?

35. Perché le routine di servizio dell'interrupt sono associate a priorità mentre le procedure normali non hanno priorità?

36. L'architettura IA-64 contiene un numero di registri insolitamente elevato, 64. La scelta di averne così tanti è legata in qualche modo all'uso dell'attribuzione di predicati? Se sì, in che modo? Se no, allora perché ce n'è così tanti?

37. Nel testo analizziamo il concetto delle `LOAD` speculative, però non si fa nessuna menzione a istruzioni di `STORE` speculative. Perché no? Perché sono essenzialmente analoghe alle `LOAD` speculative o c'è un'altra ragione per non parlarne?

38. Quando si vogliono collegare due reti locali vi si inserisce in mezzo un computer chiamato *bridge*. Ogni pacchetto trasmesso in ciascuna delle due reti causa un interrupt sul bridge, affinché possa stabilire se deve inoltrare il pacchetto da una parte all'altra. Si supponga che ci vogliono 250 μs per gestire l'interrupt e l'ispezione di ogni pacchetto, mentre il suo inoltro è a carico dell'hardware DMA e non aggrava il carico della CPU. Se tutti i pacchetti sono di 1 KB, qual è il massimo tasso di trasferimento dati che il bridge può tollerare su ciascuna delle due reti prima di cominciare a perdere pacchetti?

39. Nella Figura 5.43 il puntatore al record d'attivazione punta alla prima variabile locale. Di che informazione ha bisogno il programma per effettuare il ritorno da una procedura?

40. Si scriva una subroutine in linguaggio assemblativo per convertire in ASCII un intero binario con segno.

41. Si scriva una subroutine in linguaggio assemblativo per convertire una formula infissa in notazione polacca inversa.

42. Le torri di Hanoi non sono il solo problema ricorsivo popolare tra gli informatici. Un altro problema tra i più amati è il calcolo di *n*!. Si scriva una procedura in un linguaggio assemblativo per il calcolo di *n*!.

43. Se non si è convinti che la ricorsione è alle volte indispensabile si provi a programmare le torri di Hanoi senza usare la ricorsione e senza simulare la soluzione ricorsiva mantenendo lo stack in un array. Si tenga presente, però, che probabilmente non si riuscirà a trovare la soluzione.

Capitolo 6

Livello macchina del sistema operativo

Il "motivo conduttore" di questo libro è un calcolatore moderno visto come una serie di livelli, ciascuno responsabile di nuove funzionalità rispetto a quelli sottostanti. Abbiamo già incontrato il livello logico digitale, il livello di microarchitettura e quello di architettura dell'insieme d'istruzioni. È ora tempo di salire di un altro livello, nel dominio del sistema operativo.

Un **sistema operativo** è un programma che, dal punto di vista del programmatore, aggiunge una moltitudine di nuove istruzioni e caratteristiche di più alto livello rispetto a quelle fornite dal livello ISA. In genere il sistema operativo è implementato per la maggior parte via software, ma non esiste alcuna argomentazione teorica che ne vieti l'implementazione in hardware, come succede normalmente per i microprogrammi (quando presenti). Per brevità, ci riferiremo al livello che implementa il sistema operativo con il termine di livello **OSM** (**Operating System Machine**). Si veda al proposito la Figura 6.1.

Nonostante entrambi i livelli OSM e ISA siano astratti (nel senso che non sono veri livelli hardware), sono sostanzialmente differenti. L'insieme d'istruzioni del livello OSM contiene tutte le istruzioni disponibili ai programmatori di applicazioni, pressoché tutte le istruzioni del livello ISA, così come l'insieme delle nuove istruzioni aggiunte dal sistema operativo, dette **chiamate di sistema** (*system call*): una chiamata di sistema invoca un predeterminato servizio del sistema operativo, a tutti gli effetti una sua istruzione. Una chiamata di sistema tipica è la lettura di dati da un file. Nel prosieguo del testo indichiamo i nomi delle chiamate di sistema con un carattere che si distingue da quello del testo.

Figura 6.1 Livello macchina del sistema operativo.

Il livello OSM è sempre interpretato. Quando un utente esegue un'istruzione OSM, come la lettura di dati da file, il sistema operativo svolge l'istruzione passo dopo passo, proprio come un microprogramma svolgerebbe un'istruzione ADD. D'altra parte, quando un programma esegue un'istruzione di livello ISA, questa viene calcolata direttamente dal sottostante livello microarchitetturale, senza alcun intervento da parte del sistema operativo.

In questo libro non possiamo che fornire un'introduzione molto succinta alla materia dei sistemi operativi, ma ci soffermeremo su tre argomenti importanti: il primo è la memoria virtuale, una tecnica messa a disposizione da molti sistemi operativi moderni per far sembrare che la macchina abbia più memoria di quanta ne ha davvero; il secondo è l'I/O su file, un concetto più ad alto livello delle istruzioni di I/O, analizzato nel capitolo precedente; il terzo argomento è il calcolo parallelo, ovvero il modo in cui i processi vengono eseguiti, come comunicano e si sincronizzano. Il concetto di processo è decisamente importante, e ne daremo una descrizione dettagliata nel corso del capitolo. Per il momento potete pensare a un processo come a un programma in esecuzione unito alle sue informazioni di stato (memoria, registri, program counter, stato dell'I/O, e così via). Dopo aver analizzato questi principi in generale, vedremo come si applicano ai sistemi operativi di due delle nostre macchine paradigmatiche, cioè il Pentium 4 (Windows XP) e l'UltraSPARC III (UNIX). L'8051 non ha sistema operativo poiché è usato di norma all'interno di sistemi integrati.

6.1 Memoria virtuale

All'inizio dell'era informatica, le memorie dei computer erano costose e poco capienti. L'IBM 650, il calcolatore scientifico più avanzato del suo tempo (fine anni '50), aveva solo 2000 parole di memoria. Uno dei primi compilatori ALGOL 60 venne scritto per un computer con soltanto 1024 parole di memoria. Uno dei primi sistemi multiutente a condivisione di tempo girava abbastanza bene su di un PDP-1 con una memoria di 4096 parole di 18 bit, usata sia dal sistema operativo sia dai programmi degli utenti. In quel periodo, i programmatori impiegavano molto tempo nel tentativo di "assottigliare" i programmi perché entrassero in memorie minuscole. Alle volte si rendeva indispensabile usare un algoritmo molto più lento rispetto a un altro per il solo motivo che quello più veloce era troppo "grande", ovvero nessun programma che lo usasse avrebbe potuto essere contenuto nella memoria del calcolatore.

La soluzione tradizionale a questo problema era l'uso di una memoria secondaria, per esempio di un disco. Il programmatore divideva il programma in un certo numero di pezzi, detti **overlay** ("ricoprimento"), ciascuno dei quali poteva entrare in memoria. Al fine di eseguire il programma si recuperava il primo overlay e lo si eseguiva per un po'. Al termine della sua esecuzione il primo pezzo leggeva l'overlay successivo e lo mandava in esecuzione, e così via. Stava al programmatore ripartire il programma in parti, decidere dove memorizzarle in memoria secondaria, organizzare il trasferimento degli overlay tra la memoria principale e la secondaria, e quindi gestire l'intero procedimento di overlay senza ricevere alcun aiuto dal calcolatore.

Pur se usata per molti anni, questa tecnica richiedeva troppo lavoro da dedicare alla gestione degli overlay. Nel 1961 un gruppo di ricercatori di Manchester, in Inghilterra, propose un metodo per eseguire il processo di overlay automaticamente, in modo tale che il programmatore non avrebbe dovuto neanche accorgersi della sua presenza (Fotheringham, 1961).

Questo metodo, oggi chiamato **memoria virtuale**, presentava il vantaggio evidente di liberare il programmatore da un grosso lavoro di noiosa preparazione. Inizialmente utilizzata nell'ambito di progetti di ricerca sulla progettazione di sistemi di calcolo, a partire dagli anni '70 la memoria virtuale era divenuta disponibile su quasi tutti i calcolatori. Oggi anche i computer su chip singolo, tra cui il Pentium 4 e l'UltraSPARC III, dispongono di sistemi di memoria virtuale molto sofisticati, come avremo modo di osservare nel corso di questo capitolo.

6.1.1 Paginazione

L'idea proposta dal gruppo di Manchester era quella di separare i concetti di spazio degli indirizzi e di locazioni di memoria. Si consideri, per esempio, un computer tipico di quel momento storico, dotato plausibilmente d'istruzioni con campi d'indirizzo di 16 bit e di 4096 parole di memoria. Un programma per questo computer avrebbe potuto indirizzare 65536 parole di memoria, poiché esistono 65536 (2^{16}) stringhe di 16 bit, ciascuna corrispondente a un diverso indirizzo di memoria. Si tenga presente che il numero di parole indirizzabili dipende esclusivamente dal numero di bit nell'indirizzo e non ha nulla a che fare con il numero di parole di memoria effettivamente disponibili. Lo **spazio degli indirizzi** di questo computer è fatto dai numeri 0, 1, 2, ..., 65535 perché è quello l'insieme degli indirizzi possibili. Il computer può avere comunque meno di 65536 parole di memoria.

Prima dell'invenzione della memoria virtuale, si guardava diversamente agli indirizzi minori di 4096 e a quelli uguali o maggiori di 4096. Anche se non venivano chiamati esplicitamente così, il primo gruppo d'indirizzi era considerato spazio utile degli indirizzi, il secondo spazio inutile degli indirizzi (inutile perché non aveva corrispondenza con gli effettivi indirizzi di memoria). Non si distingueva tra spazio degli indirizzi e indirizzi di memoria, perché l'hardware imponeva una corrispondenza uno a uno tra di loro.

L'idea di separare lo spazio degli indirizzi e gli indirizzi di memoria è la seguente: in ogni istante sono direttamente accessibili 4096 parole di memoria, ma non c'è bisogno che corrispondano agli indirizzi di memoria tra 0 e 4095. Per esempio, potremmo "comunicare" al computer che d'ora in poi, ogniqualvolta ci si riferisce all'indirizzo 4096, occorre usare la parola di memoria all'indirizzo 0. Quando si referenzia l'indirizzo 4097, si deve usare la parola di memoria all'indirizzo 1; se si referenzia l'indirizzo 8191, bisogna usare la parola di memoria all'indirizzo 4095, e così via. In altri termini, abbiamo definito una corrispondenza tra lo spazio degli indirizzi e gli effettivi indirizzi di memoria, illustrata nella Figura 6.2.

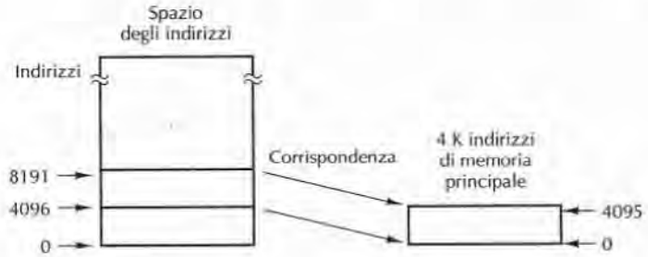


Figura 6.2 Corrispondenza tra gli indirizzi virtuali da 4096 a 8191 e gli indirizzi di memoria principale da 0 a 4095.

Secondo questa corrispondenza tra lo spazio degli indirizzi e le reali locazioni di memoria, una macchina con 4 KB di memoria, ma senza memoria virtuale, presenta una semplice corrispondenza fissa tra gli indirizzi compresi tra 0 e 4095 e le 4096 parole di memoria. Una domanda interessante è: "che cosa succede se un programma salta a un indirizzo tra 8192 e 12287?". Su di una macchina priva di memoria virtuale il programma causerebbe una trap che visualizzerebbe sullo schermo un messaggio adeguatamente severo, per esempio "memoria referenziata inesistente", e terminerebbe il programma. Su di una macchina con memoria virtuale si verificherebbero i seguenti avvenimenti:

1. il contenuto della memoria principale verrebbe salvato su disco
2. verrebbero localizzate su disco le parole dalla 8192 alla 12287
3. le parole dalla 8192 alla 12287 verrebbero caricate in memoria
4. si modificherebbe la mappa degli indirizzi per mappare gli indirizzi da 8192 a 12287 nelle locazioni di memoria da 0 a 4095
5. l'esecuzione continuerebbe come se non fosse successo nulla di insolito.

Questa tecnica di overlay automatico si chiama **paginazione** e le parti di programma lette dal disco si chiamano **pagine**.

È possibile anche una modalità più sofisticata di corrispondenza tra lo spazio degli indirizzi e i reali indirizzi di memoria. Per enfatizzare la distinzione, ci riferiremo agli indirizzi utilizzabili dal programma con il termine di **spazio degli indirizzi virtuali**, mentre ci riferiremo agli indirizzi effettivi, fisicamente cablati in memoria come allo **spazio degli indirizzi fisici**. Una **mappa di memoria** o **tabella delle pagine** specifica qual è l'indirizzo fisico corrispondente a ogni indirizzo virtuale. Facciamo qui l'ipotesi che ci sia spazio sufficiente sul disco per memorizzare l'intero spazio degli indirizzi virtuali (o quanto meno la sua porzione in uso).

I programmi vengono scritti come se ci fosse memoria sufficiente per l'intero spazio degli indirizzi virtuali, anche se non è così. I programmi possono caricare o memorizzare ogni parola nello spazio degli indirizzi virtuali, o saltare a qualsiasi istruzione situata ovunque nello spazio degli indirizzi virtuali, senza curarsi del fatto che in realtà non c'è sufficiente memoria fisica. In effetti, il programmatore può scrivere programmi senza neanche sapere che esiste la memoria virtuale, semplicemente pensando che il calcolatore abbia una grande memoria.

Questo concetto è di cruciale importanza e sarà poi messo a confronto con la segmentazione, che invece richiede che il programmatore sia conscio dell'esistenza dei segmenti. Lo sottolineiamo ancora una volta: la paginazione dà l'illusione al programmatore di una memoria grande, continua e lineare, delle stesse dimensioni dello spazio degli indirizzi virtuali. In realtà la memoria principale disponibile potrebbe essere più piccola (o più grande) dello spazio degli indirizzi virtuali. La simulazione di questa grande memoria principale mediante la paginazione non può essere individuata dal programma (se non eseguendo test sui tempi di esecuzione). Ogniqualvolta viene referenziato un indirizzo, l'istruzione o la parola di dati corrispondente sono percepite come presenti. Dal momento che il programmatore può lavorare come se la paginazione non esistesse, il meccanismo di paginazione si dice **trasparente**.

L'idea che un programmatore possa usare alcune caratteristiche inesistenti senza doversi preoccupare di come funzionano non è, dopo tutto, un'assoluta novità. L'insieme d'istruzioni a livello ISA include spesso un'istruzione `MUL`, anche se la microarchitettura sottostante manca di un moltiplicatore hardware. L'illusione che la macchina possa moltiplicare è sorretta dal microcodice. Allo stesso modo, la memoria virtuale messa a disposizione dal sistemo operativo può fornire l'illusione che gli indirizzi virtuali si riflettano nella memoria reale, anche se non è così. Solo i programmatori del sistema operativo (e gli studenti della materia) devono sapere su che cosa si regge questa illusione.

6.1.2 Implementazione della paginazione

Un requisito fondamentale per la memoria virtuale è disporre di un disco che contenga l'intero programma e tutti i dati. Concettualmente è più semplice pensare alla copia del programma su disco come all'originale, e ai pezzi trasferiti di quando in quando in memoria centrale come alle copie, piuttosto che il contrario. Naturalmente è importante mantenere l'originale aggiornato: quando vengono effettuati cambiamenti sulla copia in memoria centrale questi dovrebbero essere propagati all'originale (prima o poi).

Lo spazio degli indirizzi virtuali è suddiviso in un certo numero di pagine della stessa dimensione. Le dimensioni di pagina più in voga vanno dai 512 ai 64 KB, anche se occasionalmente vengono usate pagine di 4 MB. La dimensione di pagina è sempre una potenza di 2. Allo stesso modo, anche lo spazio degli indirizzi fisici è suddiviso in porzioni che hanno le dimensioni di una pagina, di modo che ogni pezzo di memoria principale possa contenere esattamente una pagina. Questi pezzi di memoria principale che servono a contenere le pagine si dicono **blocchi di memoria** (*page frame*). La memoria principale della Figura 6.2 contiene un solo blocco di memoria, ma una memoria reale ne contiene in genere alcune migliaia.

La Figura 6.3(a) illustra un modo di suddividere in pagine di 4 KB i primi 64 KB di spazio degli indirizzi virtuali. In realtà qui stiamo parlando di 64 KB di memoria e di pagine di 4 K d'indirizzi; un indirizzo potrebbe essere lungo un byte, ma anche una parola (nei computer in cui indirizzi consecutivi individuano parole consecutive). La memoria virtuale della Figura 6.3 potrebbe essere implementata per mezzo di una tabella delle pagine grande quanto il numero di pagine presenti nello spazio degli indirizzi virtuali. Per semplicità, nella figura mostriamo solo i primi 16 elementi della tabella. Quando un programma cerca di accedere a una parola di memoria nei primi 64 KB del suo spazio degli indirizzi virtuali (o per fare il fetch di un'istruzione o di un dato, o per salvare alcuni dati) per prima cosa genera un indirizzo virtuale compreso tra 0 e 65532 (nell'ipotesi che gli indirizzi di parola siano divisibili per 4). La generazione di questo indirizzo può avvenire tramite una qualsiasi tecnica usuale d'indirizzamento: indicizzato, indiretto, e così via.

La Figura 6.3(b) mostra una memoria fisica che consiste in otto blocchi di memoria di 4 KB. Questa memoria potrebbe essere limitata a 32 KB perché (1) è tutto ciò di cui dispone la macchina (potrebbe bastare a un processore integrato in una lavatrice o in un forno a microonde) oppure (2) perché il resto della memoria è stato allocato ad altri programmi.

Si consideri ora il meccanismo di corrispondenza tra un indirizzo virtuale di 32 bit e un indirizzo fisico della memoria principale. Dopo tutto la sola cosa comprensibile alla memoria sono gli indirizzi di memoria principale, non gli indirizzi virtuali, perciò bisogna essere in grado di fornirglieli. Ogni computer dotato di memoria virtuale è provvisto di un dispositivo atto alla corrispondenza tra indirizzi virtuali e fisici: si chiama **MMU** (*Memory Management Unit*, "unità di gestione della memoria") e può risiedere nel chip della CPU oppure in un chip separato che lavori a stretto contatto con la CPU. La MMU del nostro

Pagina	Indirizzi virtuali
15	61440 – 65535
14	57344 – 61439
13	53248 – 57343
12	49152 – 53247
11	45056 – 49151
10	40960 – 45055
9	36864 – 40959
8	32768 – 36863
7	28672 – 32767
6	24576 – 28671
5	20480 – 24575
4	16384 – 20479
3	12288 – 16383
2	8192 – 12287
1	4096 – 8191
0	0 – 4095

(a)

Blocco di memoria	Primi 32 K di memoria principale: Indirizzi fisici
7	28672 – 32767
6	24576 – 28671
5	20480 – 24575
4	16384 – 20479
3	12288 – 16383
2	8192 – 12287
1	4096 – 8191
0	0 – 4095

(b)

Figura 6.3 (a) Primi 64 KB degli indirizzi virtuali suddivisi in 16 pagine da 4 KB. (b) Memoria principale di 32 KB suddivisa in otto blocchi di memoria di 4 KB.

esempio mappa indirizzi virtuali di 32 bit in indirizzi fisici di 15 bit, perciò ha bisogno di registri d'ingresso di 32 bit e di registri d'uscita di 15 bit.

Si veda la Figura 6.4 per capire come funziona la MMU. Quando viene inviato un indirizzo virtuale di 32 bit alla MMU, questa divide l'indirizzo in un numero di pagina virtuale di 20 bit e in un offset di 12 bit all'interno della pagina (si ricordi che le pagine del nostro esempio sono di 4 KB). Il numero di pagina virtuale è usato per indicizzare la tabella delle pagine al fine di trovare l'elemento corrispondente alla pagina referenziata. Nel caso della Figura 6.4 il numero di pagina virtuale è 3 e così viene selezionato il terzo elemento della tabella delle pagine.

Per prima cosa la MMU verifica nella tabella delle pagine se la pagina referenziata è presente in memoria centrale. Ovviamente, con 2^{20} pagine virtuali e soli otto blocchi di memoria, non tutte le pagine virtuali possono trovarsi contemporaneamente in memoria. Nello svolgere questa verifica la MMU esamina il **bit presente/assente** nell'elemento della tabella delle pagine. Nel nostro esempio il bit è a 1, a significare cha la pagina si trova correntemente in memoria.

Il passo successivo è il recupero del valore del blocco di memoria contenuto nell'elemento selezionato (6 in questo caso) e la sua copia nei 3 bit più significativi del registro d'uscita di 15 bit. Ci vogliono esattamente tre bit perché ci sono otto blocchi nella memoria fisica. Contemporaneamente a questa operazione si effettua la copia dei 12 bit meno significativi dell'indirizzo virtuale (il campo offset di pagina) all'interno dei 12 bit meno significativi del registro d'uscita, come mostrato nella figura. Questo indirizzo di 15 bit è ora pronto per essere inviato alla cache o alla memoria per la sua ricerca.

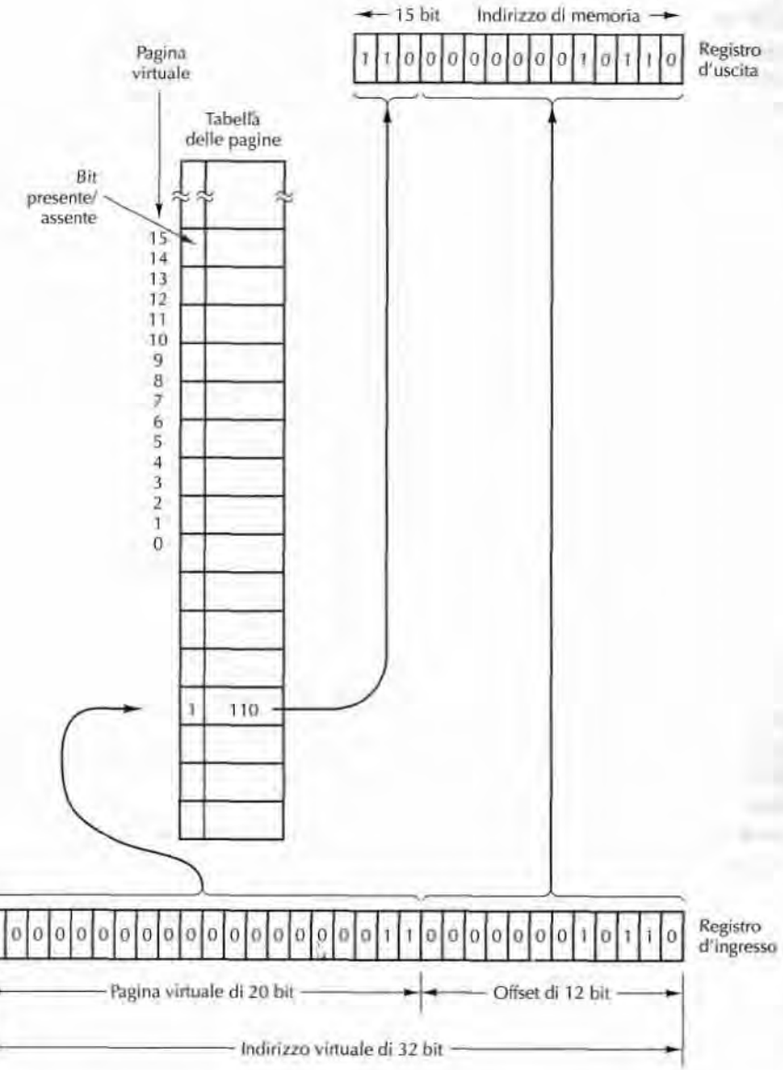


Figura 6.4 Formazione di un indirizzo fisico a partire da un indirizzo virtuale.

La Figura 6.5 mostra una possibile corrispondenza tra le pagine virtuali e i blocchi di memoria fisici. La pagina virtuale 0 si trova nel blocco di memoria 1 e la pagina virtuale 1 nel blocco di memoria 0. La pagina virtuale 2 non si trova in memoria. La pagina virtuale 3

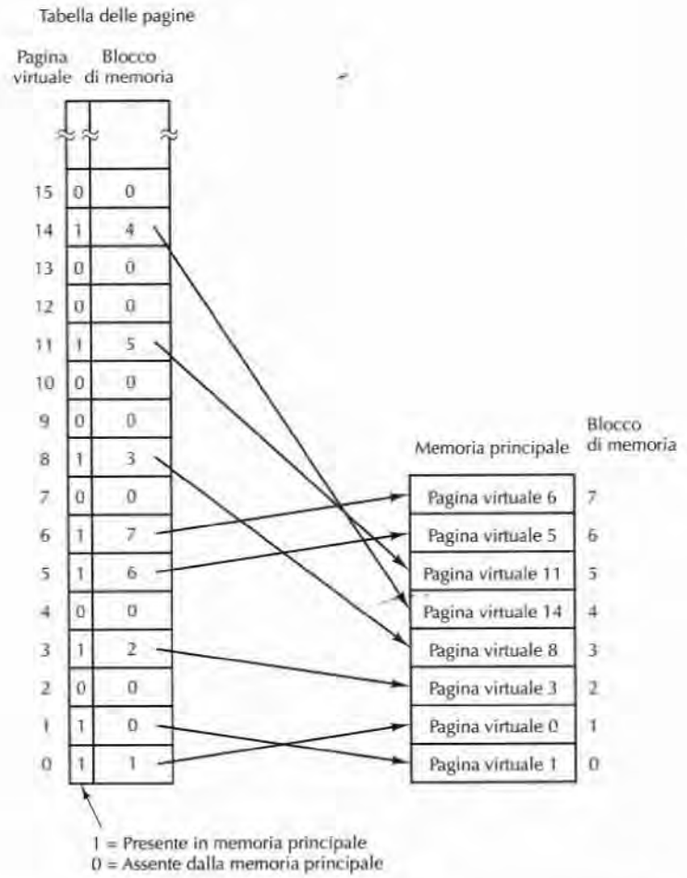


Figura 6.5 Assegnamento di 16 pagine virtuali in una memoria principale composta di otto blocchi.

si trova nel blocco di memoria 2. La pagina virtuale 4 non si trova in memoria. La pagina virtuale 5 si trova nel blocco di memoria 6, e così via.

6.1.3 Paginazione a richiesta e working set

Nella precedente trattazione abbiamo supposto che la pagina virtuale referenziata si trovasse in memoria principale, ma questa assunzione non è sempre vera, perché non c'è spazio sufficiente in memoria principale per ospitare tutte le pagine virtuali. Quando si fa riferimento a un indirizzo contenuto in una pagina che non è presente in memoria si solleva un **errore di pagina** (*page fault*). Dopo l'occorrenza di un errore di pagina è necessario che il sistema operativo legga dal disco la pagina richiesta, inserisca la sua nuova posizione all'interno della memoria fisica nella tabella delle pagine e ripeta l'istruzione che ha causato l'errore.

Su una macchina con memoria virtuale è possibile lanciare l'esecuzione di un programma anche se la memoria principale non contiene alcuna sua istruzione. Bisogna semplicemente indicare nella tabella delle pagine che ogni pagina virtuale del programma si trova in memoria secondaria e non in memoria principale. Quando la CPU prova a effettuare il fetch della prima istruzione si verifica immediatamente un errore di pagina, che causa il caricamento in memoria della pagina contenente la prima istruzione e il suo ingresso nella tabella delle pagine. Dopodiché può cominciare l'esecuzione della prima istruzione; se questa contiene due indirizzi su pagine differenti, e diverse dalla pagina dell'istruzione, allora si verificheranno altri due errori di pagina e dovranno essere recuperate altre due pagine prima che l'istruzione possa essere finalmente eseguita. L'istruzione successiva può a sua volta causare altri errori di pagina e così via.

Questo modo di funzionamento della memoria virtuale si chiama **paginazione a richiesta**, in analogia al ben noto algoritmo di allattamento a richiesta dei bambini: il neonato viene allattato quando piange (piuttosto che in base a una precisa tabella oraria). Nella paginazione a richiesta le pagine sono portate in memoria principale solo quando vengono effettivamente richieste, non prima.

L'interrogativo circa l'utilità della paginazione a richiesta è rilevante solo nella fase iniziale dell'esecuzione di un programma. Dopo un certo tempo d'esecuzione, le pagine necessarie saranno già state radunate in memoria. Se il computer è usato a condivisione di tempo e i processi si alternano ogni 100 ms, ciascun programma verrà fatto ripartire molte volte nel corso della sua esecuzione. Poiché c'è una mappa di memoria per ogni programma e questa deve essere sostituita ogniqualvolta due programmi si danno il cambio, come succede per esempio in un computer a condivisione di tempo, il problema se reiterato diventa critico.

L'approccio alternativo si basa sull'osservazione che molti programmi non referenziano il loro spazio degli indirizzi in modo uniforme, bensì i riferimenti tendono a raggrupparsi su di un piccolo numero di pagine. Questo concetto è noto come **principio di località**. Un riferimento alla memoria può servire al fetch di un'istruzione o di dati, o alla memorizzazione di dati; per ogni istante di tempo t si può definire l'insieme delle pagine usate dai k riferimenti a memoria più recenti. Denning (1968) ha chiamato questo insieme **working set** ("insieme di lavoro").

Visto che il working set in genere cambia lentamente nel tempo, è possibile indovinare con un buon margine di certezza quali saranno le pagine necessarie a un programma nel momento in cui viene fatto ripartire, basandosi sul suo working set al momento in cui è stato fermato. Così le pagine possono essere caricate in anticipo prima di far partire il programma (se ci stanno in memoria).

6.1.4 Politica di sostituzione delle pagine

In teoria il working set, l'insieme delle pagine usate attivamente e frequentemente da un programma, può essere tenuto in memoria per ridurre il numero di errori di pagina. Tuttavia, i programmatori raramente conoscono le pagine del working set, perciò l'insieme deve essere costruito dinamicamente dal sistema operativo. Quando un programma referenzia una pagina che non si trova in memoria principale questa deve essere recuperata dal disco. D'altra parte, per farle spazio sarà opportuno in genere spostare un'altra pagina sul disco e si rende quindi necessario un algoritmo per decidere quale pagina rimuovere.

Effettuare questa scelta a caso è quasi sicuramente una cattiva idea. Se capitasse di scegliere la pagina appena caricata a seguito dell'errore, un altro errore di pagina si verificherebbe non appena venisse richiesto il fetch dell'istruzione successiva. Gran parte dei sistemi operativi cerca di predire quale sia la pagina di memoria meno utile, ossia la cui mancanza produrrebbe meno effetti deleteri sull'esecuzione del programma. Un modo di agire è predire quando avverrà il successivo riferimento a ciascuna pagina e rimuovere quella per cui la predizione è più lontana nel futuro. In altre parole, si cerca di scegliere una pagina che non verrà usata per molto tempo invece di rimuoverne una che sarà richiesta a breve.

Un algoritmo molto diffuso, detto **LRU** (*Least Recently Used*), estromette la pagina usata meno recentemente perché a questa corrisponde la massima probabilità di non far parte del working set corrente. Nonostante LRU funzioni mediamente bene, ci sono situazioni patologiche, come quella che ora descriviamo, in cui fallisce miseramente.

Si immagini un programma che esegue un grosso ciclo che si estende per nove pagine virtuali e che gira su di una macchina con spazio in memoria fisica per sole otto pagine. La Figura 6.6(a) mostra la memoria principale nel momento in cui il programma raggiunge la pagina 7. Quando si tenta di fare il fetch di un'istruzione della pagina virtuale 8 si verifica un errore di pagina e si pone il problema di quale pagina estromettere. L'algoritmo LRU sceglie la pagina virtuale 0 perché è quella usata meno recentemente. La pagina virtuale 0 viene rimossa e la pagina virtuale 8 la rimpiazza, come illustrato nella Figura 6.6(b).

Dopo aver eseguito le istruzioni della pagina virtuale 8, il programma salta all'inizio del ciclo, alla pagina virtuale 0. Anche questo passo causa un errore di pagina. La pagina virtuale 0, quella appena estromessa, viene riportata in memoria. L'algoritmo LRU sceglie di rimuovere ora la pagina 1, conducendo alla situazione della Figura 6.6(c). Il programma si intrattiene per un po' nella pagina 0, quindi cerca di recuperare un'istruzione della pagina virtuale 1, causando un errore di pagina. Bisogna riportare indietro la pagina 1 ed estromettere la pagina 2.

Dovrebbe essere chiaro che in questo caso l'algoritmo LRU opera sempre la scelta peggiore (esistono anche altri algoritmi che falliscono in condizioni analoghe). Se però la memoria centrale disponibile eccede la dimensione del working set, allora LRU tende a minimizzare il numero di errori di pagina.

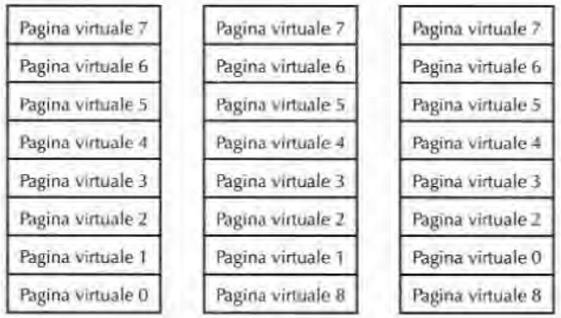


Figura 6.6 Fallimento dell'algoritmo LRU.

Un altro algoritmo di sostituzione di pagina è **FIFO** (*First-In First-Out*), che rimuove la pagina caricata meno recentemente, indipendentemente dal momento in cui sia stata referenziata. Ogni pagina è associata a un contatore, inizialmente posto a 0. Dopo la gestione di un errore di pagina, il contatore di ciascuna pagina presente in memoria viene incrementato di un'unità e il contatore della pagina appena immessa è posto a 0. Quando si rende necessario scegliere un pagina da estromettere, si seleziona quella con contatore più alto, cioè la pagina che ha assistito a più errori di pagina e che quindi è stata caricata prima di qualsiasi altra pagina presente in memoria, il che la rende (si spera) probabilmente meno utile delle altre.

Se il working set è più grande del numero di blocchi di memoria disponibili, non esiste alcun algoritmo, a meno che non abbia doti divinatorie, che possa dare buoni risultati e si verificheranno frequenti errori di pagina. Se un programma genera frequentemente e continuamente errori di pagina, allora è detto **thrashing** ("sconfitto, battuto"). Viceversa, se un programma usa una grande porzione dello spazio degli indirizzi virtuali ma ha un working set piccolo, che cambia lentamente nel tempo e che sta tutto in memoria principale, allora darà ben pochi problemi. Questa osservazione è vera anche per quei programmi che, nel corso della loro vita, usano un numero di parole della memoria virtuale centinaia di volte maggiore del numero di parole disponibili in memoria principale.

Se la pagina che sta per essere estromessa non è stata mai modificata dal momento in cui è stata caricata (cosa molto probabile se la pagina contiene soltanto istruzioni), non è necessario riscriverla su disco, perché questo ne contiene una copia fedele. Se invece è stata modificata, la copia su disco non è più accurata e la pagina deve essere riscritta.

Se esistesse un modo per stabilire quando una pagina è rimasta invariata dal suo caricamento (e in tal caso la pagina si dice "pulita", "intatta") o quando ha subito dei cambiamenti (e allora si dice "sporca"), si eviterebbero tutte le riscritture di pagine pulite, risparmiando molto tempo. Molti calcolatori dispongono nella MMU di 1 bit per pagina, detto appunto *dirty bit*, posto a 0 quando una pagina viene caricata e asserito dal microprogramma o dall'hardware quando è fatta oggetto di salvataggi (cioè quando è stata "sporcata"). Il sistema operativo può capire se una pagina è pulita o sporca osservando il suo dirty bit, e così decidere se va riscritta o meno.

6.1.5 Dimensione di pagina e frammentazione

Nell'eventualità in cui il programma e i dati dell'utente riempiano esattamente un numero intero di pagine, quando si trovano in memoria non causeranno alcuno spreco di spazio, mentre in caso contrario l'ultima pagina conterrà dello spazio inutilizzato. Per esempio, se il programma e i dati richiedono 26.000 byte su una macchina con pagine di 4096 byte, le prime sei pagine verranno riempite per un totale di 6 × 4096 = 24.576 byte, mentre l'ultima pagina conterrà 26.000 − 24.576 = 1424 byte. Dal momento che in ogni pagina c'è spazio per 4096 byte, 2672 byte verranno sprecati. Ogni volta che la settima pagina si trova in memoria questi byte "vuoti" ne occuperanno una parte senza però svolgere alcuna funzione. Il problema legato a questo tipo di spreco di byte si chiama **frammentazione interna** (perché lo spazio inutilizzato si trova all'interno di una pagina).

Se la dimensione di pagina è di n byte, la quantità media di spazio sprecato nell'ultima pagina per frammentazione interna sarà di $n/2$ byte, il che sembrerebbe suggerire l'adozione di dimensioni di pagina più piccole per minimizzare lo spreco. D'altro canto a una dimensione di pagina piccola corrisponde un gran numero di pagine e quindi un tabella delle pa-

gine più grande. Se la tabella delle pagine è implementata in hardware, le sue dimensioni richiedono un maggior numero di registri per immagazzinarla, il che eleva i costi di produzione del calcolatore. Inoltre, anche i tempi di caricamento e salvataggio dei registri aumentano ogni volta che un programma viene fatto partire o viene fermato.

D'altra parte, le pagine piccole fanno un uso poco efficiente della larghezza di banda del disco: visto che bisogna aspettare sempre almeno 10 ms prima che un trasferimento possa cominciare (tempo di ricerca più ritardo rotazionale), è evidente che i trasferimenti sono tanto più efficienti quanto più byte trasferiscono. Trasferire 8 KB aggiunge solo 0,7 ms rispetto al trasferimento di 1 KB su una macchina con velocità di trasferimento di 10 MB/s.

Tuttavia, le pagine piccole presentano il vantaggio di causare meno thrashing, essendo il working set costituito da un gran numero di piccole regioni separate dello spazio degli indirizzi virtuali. Si consideri per esempio una matrice A di 10.000 × 10.000 elementi, memorizzata come A [1,1], A [2,1], A [3,1], e così via, in parole consecutive di 8 byte. L'ordinamento per colonna implica che gli elementi della riga 1, A [1,1], A [1,2], A [1,3], e così via, si trovano distanti 80.000 byte. Se un programma svolgesse i propri calcoli su tutti gli elementi di questa riga, dovrebbe utilizzare 10.000 regioni, ciascuna separata dalla successiva da 79.992 byte. Se la dimensione di pagina è di 8 KB ciò richiede una disponibilità totale di 80 MB per contenere tutte le pagine in uso.

La dimensione di pagina di 1 KB richiederebbe invece soli 10 MB di RAM per contenere tutte le pagine. Se la memoria a disposizione è di 32 MB, il programma andrebbe in thrashing con pagine di 8 KB, ma non con pagine di 1 KB. Tutto considerato, la tendenza attuale si orienta verso dimensioni di pagina grandi.

6.1.6 Segmentazione

La memoria virtuale vista fin qui è unidimensionale perché gli indirizzi virtuali vanno da 0 a un certo indirizzo massimo, in ordine progressivo. In molte situazioni può essere decisamente preferibile disporre di due o più spazi degli indirizzi virtuali separati piuttosto che di uno solo. Per fare un esempio, un compilatore potrebbe avere molte tabelle che si riempiono man mano che la compilazione procede, tra cui:

1. la tabella dei simboli per contenere i nomi e gli attributi delle variabili
2. il codice sorgente, memorizzato per la visualizzazione del listato
3. una tabella per contenere tutte le costanti intere e in virgola mobile utilizzate
4. l'albero sintattico del programma
5. lo stack, utilizzato per le chiamate di procedura del compilatore.

Ciascuna di queste tabelle cresce continuamente durante la compilazione. L'ultima addirittura cresce e decresce in modi imprevedibili. In una memoria unidimensionale queste cinque tabelle sarebbero state allocate in unità contigue dello spazio degli indirizzi virtuali, come illustrato nella Figura 6.7.

Se un programma ha un numero eccezionalmente elevato di variabili, la parte d'indirizzi allocata per la tabella dei simboli si potrebbe riempire, nonostante ci sia molto spazio libero nelle altre tabelle. Naturalmente il compilatore potrebbe cavarsela con un messaggio di errore comunicando l'impossibilità di proseguire la compilazione per la presenza di troppe

Figura 6.7 In uno spazio degli indirizzi unidimensionale due tabelle in espansione possono collidere.

variabili, ma non è un comportamento molto leale, specie considerando tutto lo spazio libero delle altre tabelle.

Un'altra possibilità prevede che il compilatore si comporti come Robin Hood, sottraendo spazio alle tabelle che ne hanno troppo per ridistribuirlo a quelle che ne sono povere. Questo rimescolamento è possibile, ma è un po' come gestire manualmente gli overlay: se va bene è una seccatura, se va male diventa un grosso carico di lavoro tedioso e poco remunerativo.

C'è bisogno di un modo per liberare il programmatore dalla gestione dell'espansione e della contrattura delle tabelle, analogamente al modo in cui la memoria virtuale elimina la preoccupazione di dover organizzare i programmi in overlay.

Una soluzione semplice è prevedere molti spazi degli indirizzi completamente indipendenti, detti **segmenti**. Ogni segmento consiste in una sequenza lineare d'indirizzi, che va da 0 a un valore massimo. La lunghezza di ciascun segmento può variare a sua volta da 0 a un massimo consentito: segmenti diversi potrebbero avere, e in genere hanno, lunghezze diverse. Per di più la lunghezza dei segmenti potrebbe cambiare durante l'esecuzione: la lunghezza del segmento dello stack potrebbe essere incrementata dopo ogni push e decrementata dopo ogni pop.

Poiché ciascun segmento costituisce uno spazio separato d'indirizzi, i diversi segmenti possono crescere o decrescere indipendentemente, senza influenzarsi a vicenda. Se uno stack ha bisogno di più spazio per crescere all'interno del proprio segmento, allora può prenderselo perché non c'è nient'altro nel suo spazio degli indirizzi con cui possa collidere. Ovviamente può sempre succedere che un segmento venga riempito completamente, ma accade molto raramente, perché la sua dimensione è di solito molto grande. Per specificare un indirizzo di una memoria segmentata o bidimensionale, il programma deve fornire un indirizzo composto da due parti: un numero di segmento e un indirizzo all'interno del segmento. La Figura 6.8 illustra una memoria segmentata usata per le tabelle del compilatore illustrate in precedenza.

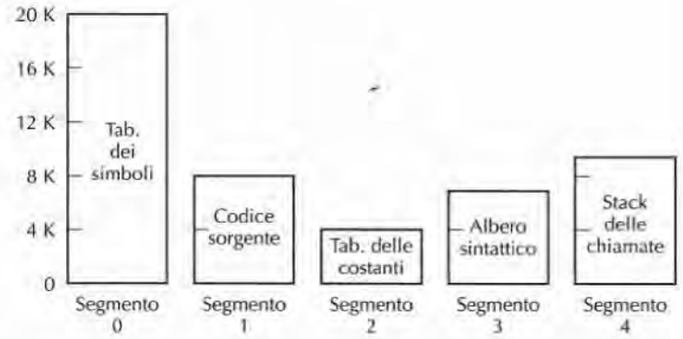


Figura 6.8 Una memoria segmentata consente a ciascuna tabella di crescere o decrescere indipendentemente dalle altre.

Sottolineiamo il fatto che un segmento è un'entità *logica* della cui esistenza il programmatore è conscio e quindi la usa come una singola entità logica. Un segmento potrebbe contenere una procedura, un array, uno stack o una raccolta di variabili e in genere contiene oggetti di tipo omogeneo.

Una memoria segmentata presenta anche altri vantaggi oltre a semplificare la gestione delle strutture dati con dimensione variabile. Se ogni procedura occupa un segmento separato e inizia all'indirizzo 0, il collegamento delle procedure compilate separatamente risulta grandemente agevolato. Quando tutte le procedure di un programma sono state compilate e collegate, la chiamata di una procedura del segmento n userà l'indirizzo (n, 0) per indirizzare la parola 0 (il punto d'ingresso).

Se poi la procedura del segmento n viene modificata e ricompilata, non c'è bisogno di modificare le altre procedure (perché il loro indirizzo iniziale non è stato modificato), neanche nel caso in cui la nuova versione sia più grande della vecchia. In una memoria unidimensionale le procedure sono di norma impacchettate una di seguito all'altra, senza spazio tra di loro, perciò la modifica delle dimensioni di una procedura può causare la variazione dell'indirizzo iniziale di altre procedure indipendenti. La cosa implica la modifica di tutte le procedure che richiamano una di quelle spostate, per memorizzare il loro nuovo indirizzo iniziale. Questo procedimento diventa costoso su programmi composti da centinaia di procedure.

La segmentazione facilita anche la condivisione di dati o procedure tra più programmi. Se un computer ha molti programmi in esecuzione parallela (vera o simulata) che usano tutti certe procedure di libreria, fornire a ciascuno una propria copia privata delle procedure risulterebbe un gran spreco di memoria principale. Se invece ogni procedura occupa un segmento separato, possono essere condivise facilmente, risparmiando così lo spazio di memoria richiesto da copie multiple.

Si è detto che ogni segmento forma un'entità logica visibile al programmatore, come una procedura, un array o uno stack, dunque è possibile definire modalità di protezione diverse su segmenti diversi. Un segmento di procedura potrebbe essere specificato come solo eseguibile, proibendone accessi in lettura o scrittura. Un array di numeri in virgola mobile potrebbe essere impostato come accessibile in lettura/scrittura, ma non in esecuzione, così che verrebbero rilevati e negati eventuali tentativi di salto verso di esso. Questi tipi di protezione sono di grande aiuto nello scovare errori di programmazione.

Domanda	Paginazione	Segmentazione
Il programmatore deve esserne conscio?	No	Sì
Quanti spazi lineari d'indirizzi ci sono?	1	Molti
Lo spazio degli indirizzi virtuali può eccedere la dimensione della memoria?	Sì	Sì
È facile gestire le tabelle a dimensione variabile?	No	Sì
Perché è stata inventata questa tecnica?	Per simulare memorie grandi	Per fornire molteplici spazi d'indirizzi

Figura 6.9 Confronto tra paginazione e segmentazione.

È importante rendersi conto del perché la protezione è concepibile in una memoria segmentata, ma non in una memoria paginata (cioè lineare). In una memoria segmentata l'utente conosce il contenuto di ogni segmento e di norma nessun segmento contiene sia una procedura sia uno stack, ma l'una o l'altro. Il fatto che ciascun segmento contenga un solo tipo di oggetto rende possibile l'attribuzione di una protezione appropriata al tipo in questione. La Figura 6.9 mette a confronto paginazione e segmentazione.

Il contenuto di una pagina è, in un certo senso, accidentale. Il programmatore non ha alcun sentore della paginazione. Anche se in linea di principio sarebbe possibile usare una manciata di bit per ogni elemento della tabella delle pagine al fine di specificare il tipo di accesso consentito, in tal caso il programmatore dovrebbe tener traccia, nel proprio spazio degli indirizzi, dei punti di transizione tra pagine. Questo non va bene perché è proprio il genere di amministrazione che la paginazione vuole eliminare. L'utente di una memoria segmentata ha l'illusione che tutti i segmenti si trovino in memoria allo stesso tempo, perciò può referenziarli senza curarsi della loro gestione.

6.1.7 Implementazione della segmentazione

La segmentazione può essere implementata in due modi diversi: con lo swapping ("scambio") o con la paginazione. Nel primo schema, in ogni istante la memoria contiene un certo numero di segmenti. Se viene fatto un riferimento a un segmento che non si trova correntemente in memoria, allora viene caricato. Se non c'è abbastanza spazio per accoglierlo, bisogna prima scrivere uno o più segmenti su disco (a meno che esista già una copia pulita su disco, nel qual caso la copia in memoria viene semplicemente abbandonata). In un certo qual modo lo swapping di segmenti non è dissimile dalla paginazione su richiesta: i segmenti sono caricati una volta richiesti.

Tuttavia, l'implementazione della segmentazione differisce dalla paginazione in modo essenziale: le pagine hanno dimensione fissa, i segmenti no. La Figura 6.10(a) mostra un esempio di memoria fisica che contiene inizialmente cinque segmenti. Si consideri ora che cosa succede quando il segmento 1 viene estromesso e viene caricato al suo posto il segmento 7, che è più piccolo. Ci troviamo nella configurazione di memoria della Figura 6.10(b). Lo spazio tra il segmento 7 e il 2 è area non utilizzata, cioè una lacuna. Poi il segmento 4 è rimpiazzato dal segmento 5 (Figura 6.10(c)) e il segmento 3 dal 6 (Figura 6.10(d)). Dopo un po' di tempo la memoria si troverà divisa in segmenti e lacune. Questo fenomeno si chiama **frammentazione esterna** (perché lo spazio sprecato è al di fuori dei segmenti). Alle volte ci si riferisce alla frammentazione esterna con il termine di **checkerboarding**, cioè "comportamento a scacchiera".

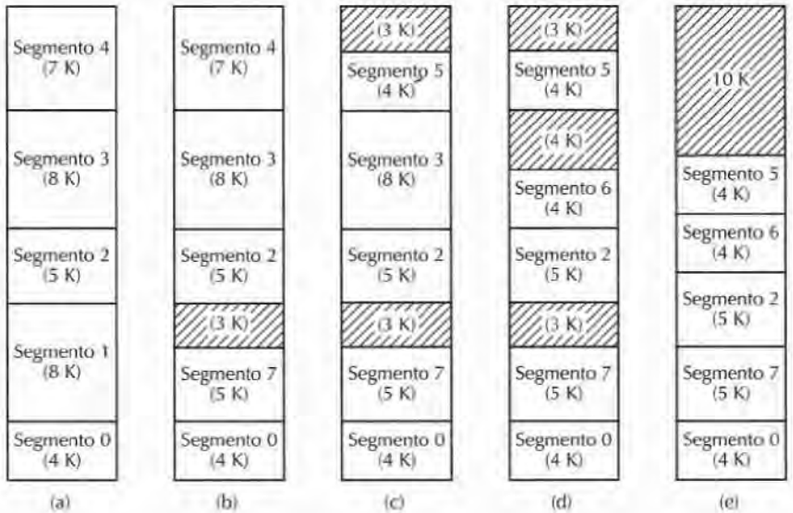


Figura 6.10 (a)-(d) Incremento della frammentazione esterna. (e) Rimozione della frammentazione esterna mediante compattamento.

Si consideri ciò che si verificherebbe se il programma accedesse al segmento 3 nel momento in cui la memoria è affetta da frammentazione esterna, come nella Figura 6.10(d). Lo spazio globalmente libero è 10 K, più di quanto serva al segmento 3, ma poiché è distribuito in frammenti piccoli e inutili, il segmento 3 non può essere caricato. Bisognerebbe rimuovere un altro segmento.

C'è un modo per evitare la frammentazione esterna: ogni volta che appare una lacuna, si può spostare il segmento che segue la lacuna in direzione della locazione di memoria 0, eliminando così quella lacuna, ma creandone una grande alla fine del segmento. In alternativa si potrebbe aspettare fino al momento in cui la frammentazione esterna diventi grave (per esempio quando eccede una certa percentuale della memoria totale) prima di eseguire un compattamento (spostando le lacune lontano dai segmenti). La Figura 6.10(e) mostra la memoria della Figura 6.10(d) dopo compattamento. Lo scopo del compattamento della memoria (a volte chiamato *garbage collection*, cioè "raccolta della spazzatura") è di compattare tute le piccole, inutilizzabili lacune in una sola, in cui poter caricare uno o più segmenti. Il compattamento presenta l'inconveniente ovvio di portar via del tempo ed eseguirlo dopo la creazione di ogni singola lacuna, consumando troppe risorse.

Se il tempo di compattamento è così lungo da risultare inaccettabile, si rende necessario trovare un algoritmo per determinare quale lacuna usare per caricare un particolare segmento. Tale algoritmo richiede il mantenimento di una lista degli indirizzi e della dimensione delle lacune. Un algoritmo diffuso, chiamato **best fit** ("miglior corrispondenza"), sceglie la lacuna più piccola che può contenere il segmento, nel tentativo di far corrispondere un segmento a ogni lacuna per evitare di frammentarne una che potrebbe servire per ospitare un segmento grande.

Un altro algoritmo molto usato, **first fit** ("prima corrispondenza"), scandisce circolarmente la lista delle lacune e sceglie la prima in grado di ospitare il segmento. Naturalmente

così facendo impiega meno tempo di quanto sia necessario alla scansione dell'intera lista operata da best fit. Sorprende invece sapere che first fit è un algoritmo migliore di best fit anche in termini di prestazioni complessive, dal momento che best fit tende a generare moltissime lacune totalmente inservibili (Knuth, 1997).

First fit e best fit tendono a diminuire la dimensione media delle lacune. Ogni volta che un segmento viene caricato in una lacuna più grande di lui, il che capita praticamente sempre (le corrispondenze esatte sono rare), la lacuna viene divisa in due parti: una parte è occupata dal segmento, e l'altra costituisce una nuova lacuna, più piccola di quella originaria. A meno di un processo di compensazione che ricrei lacune più grandi a partire da quelle piccole, alla lunga sia first fit sia best fit tendono a creare in memoria molte lacune inutilizzabili.

Un meccanismo di compensazione è il seguente. Quando viene rimosso un segmento dalla memoria e uno o entrambi i suoi vicini sono lacune e non segmenti, le lacune adiacenti possono saldarsi in una sola. Se venisse rimosso il segmento 5 dalla Figura 6.10(d), le lacune circostanti e i 4 KB usati dal segmento verrebbero fusi in una sola lacuna di 11 KB.

All'inizio di questo paragrafo si è detto che ci sono due modi di implementare la segmentazione: con lo swapping o con la paginazione. La trattazione ha fin qui riguardato lo swapping. Secondo questo schema i segmenti vengono spostati dalla memoria al disco su richiesta. L'altro modo di implementare la segmentazione è di suddividere ogni segmento in un numero prefissato di pagine e svolgere la loro paginazione su richiesta. Secondo questo schema alcune delle pagine del segmento potranno trovarsi in memoria, altre sul disco. La paginazione dei segmenti richiede una tabella delle pagine per ogni segmento. Non essendo un segmento altro che uno spazio lineare degli indirizzi, tutte le tecniche viste fin qui per la paginazione si applicano direttamente a ogni segmento. L'unica novità è la presenza di una tabella della pagine per ciascun segmento.

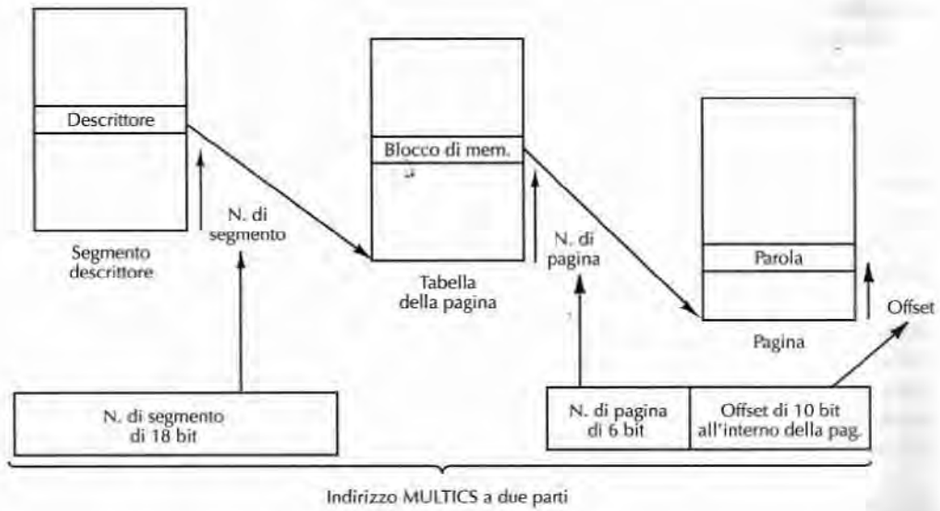


Figura 6.11 Conversione di un indirizzo MULTICS a due parti in un indirizzo fisico.

Uno dei primi sistemi operativi a combinare la segmentazione con la paginazione fu **MULTICS** (*MULTiplexed Information and Computing Service*), nato come sforzo congiunto di M.I.T., dei Laboratori Bell e della General Electric (Corbato e Vyssotsky, 1965; Organick, 1972). Gli indirizzi di MULTICS erano formati da due parti: un numero di segmento e un indirizzo all'interno del segmento. C'era un segmento descrittore per ogni processo, contenente a sua volta un descrittore per ciascun segmento. Quando l'hardware riceveva un indirizzo virtuale, il numero di segmento era usato come indice nel segmento descrittore per localizzare il descrittore del segmento cui accedere, come mostrato nella Figura 6.11. Il descrittore puntava alla tabella delle pagine, rendendo possibile la paginazione di ogni segmento nelle modalità usuali. Al fine di incrementare le prestazioni, una **memoria associativa** hardware di 16 bit era usata per contenere le combinazioni segmento/pagina usate più di recente, in modo da potervi accedere velocemente. Anche se MULTICS è oggi estinto, il suo spirito gli è sopravvissuto, dato che le memorie virtuali di tutte le CPU Intel a partire dal 386 sono state modellate sul suo esempio.

6.1.8 Memoria virtuale del Pentium 4

Il Pentium 4 è fornito di un sistema sofisticato di memoria virtuale che supporta la paginazione a richiesta, la segmentazione pura e la segmentazione paginata. Nel nucleo della memoria virtuale del Pentium 4 ci sono due tabelle: **LDT** (*Local Descriptor Table*, "tabella dei descrittori locali") e **GDT** (*Global Descriptor Table*, "tabella dei descrittori globali"). Ciascun programma ha la propria LDT, ma l'unica GDT è condivisa da tutti i programmi. LDT descrive i segmenti locali a ogni programma, tra cui il suo codice, i dati, lo stack e così via, mentre GDT descrive i segmenti di sistema, compreso il sistema operativo stesso.

Come già descritto nel Capitolo 5, per accedere a un segmento un programma Pentium 4 deve caricare un selettore per quel segmento in uno dei suoi registri di segmento. Durante l'esecuzione, CS contiene il selettore per il segmento di codice, DS quello per il segmento di dati, e così via. Ogni selettore è un numero di 16 bit, come illustrato nella Figura 6.12.

Un bit del selettore specifica se il segmento è locale oppure globale (cioè se si trova in LDT o in GDT). Altri 13 bit specificano il numero dell'elemento di LDT o di GDT, perciò entrambe le tabelle possono contenere 8 KB di descrittori di segmenti. Gli altri 2 bit riguardano la protezione e verranno trattati in seguito. Il descrittore 0 non è valido e causa una trap se viene usato. È possibile caricarlo senza problemi in un registro di segmento per indicare che il registro non è al momento disponibile, ma se viene usato provoca comunque una trap.

Nell'istante in cui un selettore viene caricato in un registro di segmento, il descrittore corrispondente è recuperato da LDT o GDT e memorizzato nei registri interni della MMU,

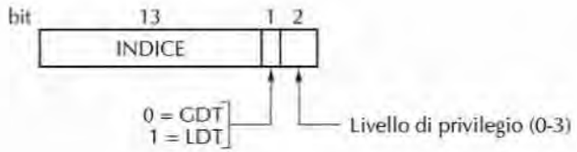


Figura 6.12 Selettore del Pentium 4.

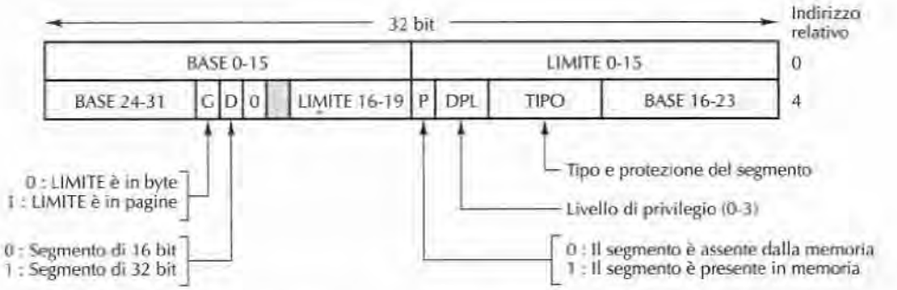


Figura 6.13 Descrittore di un segmento di codice nel Pentium 4. I segmenti di dati sono leggermente diversi.

dove può essere referenziato velocemente. Un descrittore è fatto di 8 byte, compreso l'indirizzo della base del segmento, la sua dimensione e altre informazioni (Figura 6.13).

Il formato del selettore è stato concepito in modo intelligente per facilitare la localizzazione del descrittore. Prima si sceglie tra LDT e GDT, in base al bit 2 del selettore, quindi si copia il selettore in un registro di lavoro della MMU e ne vengono azzerati i 3 bit meno significativi, il che equivale a moltiplicare per otto il numero di 13 bit del selettore. Infine gli si somma l'indirizzo della tabella LDT o GDT (tenuto nei registri interni della MMU) per ottenere un puntatore diretto al descrittore. Per esempio, il selettore 72 fa riferimento all'elemento 9 di GDT, che si trova all'indirizzo GDT + 72.

Seguiamo passo dopo passo il meccanismo di conversione di una coppia (selettore, offset) in un indirizzo fisico.

Non appena stabilito il registro di segmento da utilizzare, l'hardware può trovare il descrittore completo corrispondente al selettore memorizzato nel registro interno. Se il segmento non esiste (selettore 0), oppure se non si trova correntemente in memoria (P vale 0), si verifica una trap. La prima eventualità è un errore di programmazione, la seconda richiede l'intervento del sistema operativo per recuperare il segmento.

Dopo di ciò, l'hardware verifica se l'offset eccede la fine del segmento e, in caso affermativo, si ha un'altra trap. Secondo logica, il descrittore dovrebbe contenere semplicemente un campo di 32 bit per specificare la dimensione del segmento, ma essendoci soltanto 20 bit disponibili, si usa uno schema differente. Se il bit G di granularità vale 0, allora il campo LIMITE rappresenta la dimensione esatta del segmento, fino a un massimo di 1 MB. Se invece G vale 1, il campo LIMITE restituisce la dimensione del segmento in pagine invece che in byte. La dimensione di pagina nel Pentium 4 non è mai minore di 4 KB, perciò bastano 20 bit per segmenti grandi fino a 2^{32} byte.

Ipotizziamo che il segmento si trovi in memoria e che l'offset sia corretto. In tal caso il Pentium 4 somma il campo BASE di 32 bit del descrittore all'offset, per formare quello che è detto un **indirizzo lineare**, mostrato nella Figura 6.14. Il campo BASE è suddiviso in tre parti, sparpagliate all'interno del descrittore per motivi di compatibilità con l'80286, il cui campo BASE è di soli 24 bit. La lunghezza del campo BASE permette a un segmento di cominciare in una locazione qualsiasi dello spazio degli indirizzi lineari di 32 bit.

Se la paginazione è disabilitata (da un bit del registro globale di controllo) l'indirizzo lineare è interpretato come indirizzo fisico e viene quindi inviato alla memoria per la sua let-

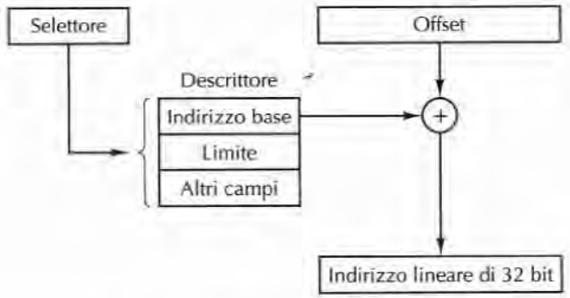


Figura 6.14 Conversione di una coppia (selettore, offset) in un indirizzo lineare.

tura o scrittura. Senza la paginazione abbiamo dunque uno schema di segmentazione pura, dove il descrittore di un segmento ne individua l'indirizzo base. È consentito ai segmenti di sovrapporsi accidentalmente, forse perché la verifica che siano tutti disgiunti richiederebbe troppo tempo e comporterebbe eccessive complicazioni.

D'altra parte, se la paginazione è abilitata, l'indirizzo lineare è interpretato come un indirizzo virtuale e viene trasformato nell'indirizzo fisico tramite le tabelle delle pagine, come già visto. La sola complicazione qui è che, con indirizzi virtuali di 32 bit e pagine di 4 KB, è possibile avere segmenti da un milione di pagine, perciò viene usata una corrispondenza a due livelli per ridurre la dimensione della tabella delle pagine nel caso di segmenti piccoli.

Ogni programma in esecuzione ha una **directory delle pagine** costituita da 1024 elementi di 32 bit e localizzata presso un indirizzo puntato da un registro globale. Ogni elemento di questa directory punta a una tabella delle pagine costituita anch'essa di 1024 elementi di 32 bit. Gli elementi della tabella delle pagine puntano ai blocchi di memoria. Lo schema è quello della Figura 6.15.

La Figura 6.15(a) raffigura un indirizzo lineare suddiviso in tre campi: `DIR`, `PAGINA` e `OFF`. Il campo `DIR` è usato per primo per indicizzare la directory delle pagine al fine di trovare il puntatore alla tabella delle pagine appropriata. Poi viene usato il campo `PAGINA` come indice nella tabella delle pagine per trovare l'indirizzo del blocco di memoria. Infine `OFF` è sommato all'indirizzo del blocco di memoria per formare l'indirizzo fisico del byte o della parola cui accedere.

Gli elementi della tabella delle pagine sono tutti di 32 bit, di cui 20 per il numero del blocco di memoria. I bit restanti contengono bit di accesso, un dirty bit (asserito dall'hardware a beneficio del sistema operativo se il blocco è stato modificato dopo il caricamento), bit di protezione e altri bit.

Ogni tabella delle pagine contiene elementi per 1024 blocchi di memoria di 4 KB, perciò una sola tabella delle pagine raggiunge 4 MB di memoria. Ai segmenti più piccoli di 4 MB basta una directory delle pagine con un solo elemento, cioè il puntatore alla sua unica tabella delle pagine. Così facendo l'informazione accessoria per i segmenti piccoli è di sole due pagine, invece dei milioni di pagine che sarebbero richieste in una tabella delle pagine a un solo livello.

Per evitare di accedere ripetutamente alla memoria, la MMU del Pentium 4 ha un hardware speciale per la ricerca veloce delle combinazioni `DIR`–`PAGINA` usate più di recente,

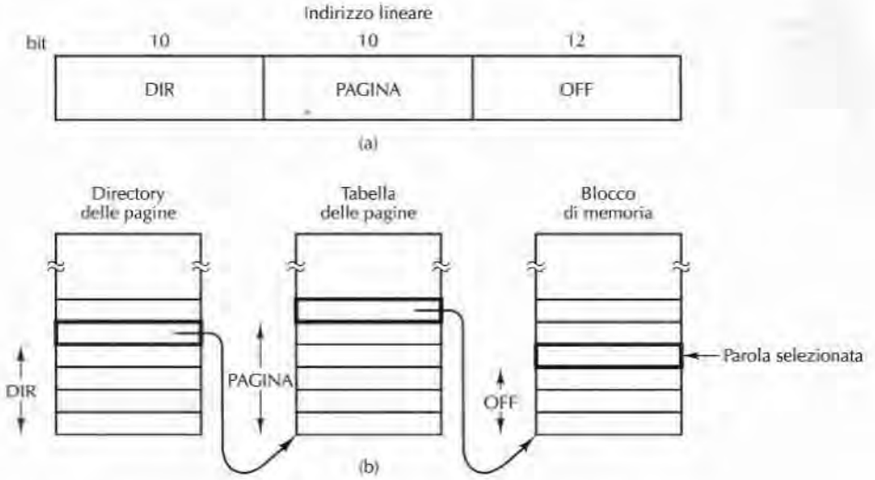


Figura 6.15 Corrispondenza tra un indirizzo lineare e un indirizzo fisico.

e le mappa negli indirizzi fisici dei blocchi di memoria corrispondenti. I passi illustrati nella Figura 6.15 vengono svolti effettivamente solo quando la combinazione corrente non sia stata usata di recente.

Non c'è alcuna ragione per avere un valore diverso da 0 nel campo `BASE` del descrittore quando si usa la paginazione. Ciò è dovuto alla semplice osservazione che l'effetto del campo `BASE` è di accedere a un elemento nel mezzo della directory delle pagine che si trova a un certo offset dall'elemento iniziale, invece che all'elemento iniziale stesso. Le uniche ragioni che giustificano l'esistenza del campo `BASE` sono consentire la segmentazione pura (non paginata) e garantire la retrocompatibilità con il vecchio 80286, privo di paginazione.

Vale la pena menzionare il fatto che, se un'applicazione non ha bisogno della segmentazione e richiede un singolo spazio degli indirizzi di 32 bit paginato, è facile soddisfare la sua richiesta. Basta impostare tutti i registri di segmento allo stesso selettore, il cui descrittore ha `BASE` = 0 e `LIMITE` = massimo. Se si usa un solo spazio degli indirizzi, l'offset dell'istruzione diviene l'indirizzo lineare e si ottiene in effetti la paginazione tradizionale.

Abbiamo così concluso la trattazione della memoria virtuale del Pentium 4. Resta da aggiungere qualche osservazione a proposito della protezione, essendo questa una materia intimamente legata alla memoria virtuale. Il Pentium 4 supporta quattro livelli di protezione, da 0 (massimi privilegi) a 3, illustrati nella Figura 6.16. In ogni momento un programma in esecuzione si trova a un certo livello di protezione, indicato da un campo di 2 bit nel registro hardware **PSW** (*Program Status Word*), contenente i codici di condizione e svariati altri bit di stato. Inoltre, ogni segmento del sistema si trova a un certo livello di protezione.

Fintantoché un programma si limita a usare segmenti al suo stesso livello, non ci sono problemi. Sono consentiti i tentativi di accesso a dati che si trovano a un livello più alto, mentre i tentativi di accesso a dati a livello più basso sono vietati e causano una trap. È consentito richiamare procedure a livelli diversi (più alti o più bassi), benché in modo severamente controllato. Un'istruzione `CALL` che voglia effettuare una chiamata tra livelli

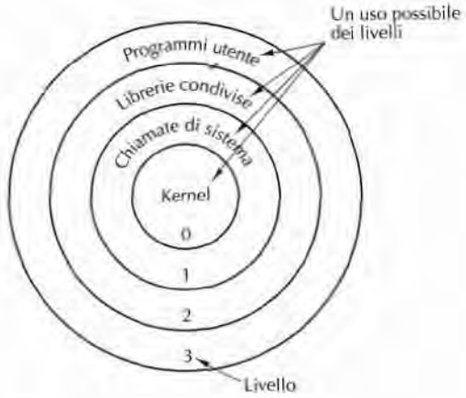


Figura 6.16 Protezione nel Pentium 4.

deve contenere un selettore invece di un indirizzo. Questo selettore designa un descrittore che si chiama **call gate** ("accesso alle chiamate") e che restituisce l'indirizzo della procedura da invocare. Così facendo si nega la possibilità di saltare nel mezzo di un segmento di codice di livello diverso; è possibile accedere solo ai punti d'ingresso ufficiali.

Un uso possibile di questo meccanismo è suggerito dalla Figura 6.16. A livello 0 troviamo il kernel del sistema operativo, che gestisce l'I/O, la memoria e altre questioni critiche. A livello 1 troviamo il gestore delle chiamate di sistema. I programmi utente possono richiamare procedure a questo livello perché svolgano chiamate di sistema, ma solo quelle appartenenti a una lista specifica di procedure protette. Il livello 2 contiene le procedure di libreria, eventualmente condivise tra molti programmi in esecuzione. I programmi utente le possono richiamare, ma non modificare. Infine i programmi utente girano a livello 3, cui è associata la protezione minima. Al pari dello schema di gestione della memoria, anche il sistema di protezione del Pentium 4 è ispirato fortemente a quello di MULTICS.

Le trap e gli interrupt usano un procedimento simile a quello dei call gate: anch'essi referenziano descrittori, invece d'indirizzi assoluti, e questi descrittori puntano alle specifiche procedure da eseguire. Il campo `TIPO` della Figura 6.13 opera la distinzione tra segmenti di codice e segmenti di dati, oltre che tra vari tipi di call gate.

6.1.9 Memoria virtuale dell'UltraSPARC III

L'UltraSPARC III è una macchina a 64 bit che supporta una memoria virtuale paginata basata su indirizzi virtuali di 64 bit. Per ragioni ingegneristiche e di costo, i programmi non possono usare l'intero spazio degli indirizzi virtuali di 64 bit, ma sono supportati soli 44 bit, perciò i programmi non possono eccedere il byte $1{,}8 \times 10^{13}$. La memoria virtuale consentita è divisa in due zone di 2^{43} byte ciascuna, una in cima, l'altra in fondo allo spazio degli indirizzi virtuali. Tra di loro c'è una zona contenente indirizzi che non possono essere usati; eventuali tentativi di accedervi causano un errore di pagina.

La memoria fisica massima ammessa su un UltraSPARC III è di 2^{41} byte, cioè circa 2200 GB, quanto basta per gran parte delle applicazioni comuni. Sono supportate quattro

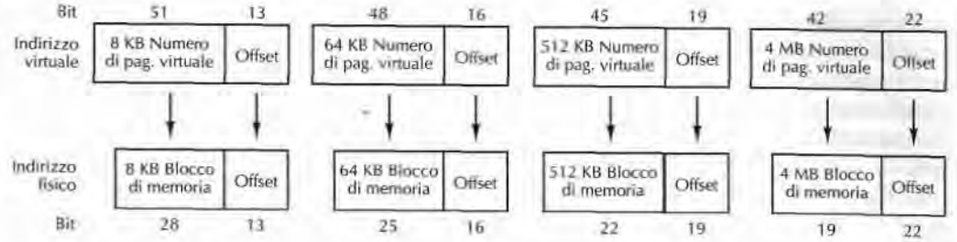


Figura 6.17 Corrispondenze tra indirizzi virtuali e fisici nell'UltraSPARC.

dimensioni di pagina: 8, 64, 512 KB e 4 MB. Le corrispondenze indotte da queste quattro dimensioni di pagina sono illustrate nella Figura 6.17.

A causa della notevole grandezza del suo spazio degli indirizzi virtuali, l'UltraSPARC III non ha una tabella delle pagine semplice come quella del Pentium 4, perché non sarebbe pratica. Invece la sua MMU è costretta a usare un approccio molto diverso: contiene una tabella hardware chiamata **TLB** (*Translation Lookaside Buffer*), una tabella associativa che mappa i numeri di pagina virtuali in numeri di blocchi di memoria fisici. Nel caso delle dimensioni di pagina di 8 KB ci sono 2^{31} numeri di pagina virtuale, ossia più di 2 miliardi. Ovviamente non tutte possono essere mappate, la TLB contiene solo i numeri di pagina virtuale più recentemente usati.

La tracciatura delle istruzioni e dei dati avviene separatamente: la TLB contiene per ciascuna delle due categorie i 64 numeri di pagina virtuale usati più di recente. Ogni elemento di TLB contiene un numero di pagina virtuale e il numero del blocco di memoria fisico corrispondente. Alla ricezione di un numero di processo, detto **contesto**, e di un indirizzo virtuale all'interno di quel contesto, la MMU usa una circuiteria speciale per confrontare il numero di pagina virtuale in esso contenuto con tutti gli elementi di TLB simultaneamente, al fine di trovare quel contesto. Se l'esito del confronto è positivo, si forma un indirizzo fisico di 41 bit, componendo il numero del blocco di memoria estratto da TLB con l'offset preso dall'indirizzo virtuale; in questa fase si impostano anche alcuni flag, compresi i bit di protezione. La TLB è illustrata nella Figura 6.18(a).

Se invece non si trova alcuna corrispondenza, si verifica un **fallimento di TLB**, che causa una trap rivolta al sistema operativo, cui spetta la gestione dell'errore. Si tenga presente che un fallimento di TLB è molto diverso da un errore di pagina. Un fallimento di TLB si può verificare anche se la pagina cui si fa riferimento è ancora in memoria, e in linea teorica il sistema operativo è libero di scegliere come agire per caricare in TLB un nuovo elemento con la pagina virtuale richiesta. A ogni modo si tratta di un'operazione cruciale e l'hardware viene in aiuto del software per velocizzare le operazioni, ammesso che il software intenda collaborare.

In particolare, il sistema operativo dovrebbe gestire in memoria una cache software degli elementi di TLB molto usati, detta **TSB** (*Translation Storage Buffer*, "buffer per la memorizzazione delle traduzioni"). Questa tabella è organizzata come una cache delle pagine virtuali a corrispondenza diretta: ogni elemento di 16 byte di TSB fa riferimento a una pagina virtuale e contiene un bit di validità, il numero di contesto, l'etichetta dell'indirizzo virtuale, il numero di pagina fisica e altri flag. Se la dimensione della cache è di, poniamo, 8192

elementi, allora competono per essere inserite nell'elemento 0 di TSB tutte le pagine virtuali i cui 13 bit meno significativi vengono mappati in 0000000000000. Analogamente competono per essere inserite nell'elemento 1 di TSB tutte le pagine virtuali i cui 13 bit meno significativi vengono mappati in 0000000000001, come mostrato nella Figura 6.18(b). La dimensione di TSB è stabilita via software e viene comunicata alla MMU tramite registri speciali, accessibili solo dal sistema operativo.

Dopo un fallimento di TLB, il sistema operativo verifica se l'elemento corrispondente in TSB contiene la pagina virtuale richiesta. A questo punto la MMU collabora calcolando l'indirizzo di questo elemento e rendendolo disponibile in un suo registro accessibile dal sistema operativo. Se la pagina virtuale è in TSB, un elemento di TLB viene epurato per consentire la copia dell'elemento di TSB richiesto. Anche l'hardware collabora scegliendo l'elemento di TLB da eliminare tramite l'algoritmo LRU a 1 bit.

Se il processo non ha esito positivo e la pagina virtuale richiesta non è nella cache, il sistema operativo usa un'altra tabella per localizzare le informazioni sulla pagina (che potrebbe trovarsi in memoria o su disco). La tabella usata per questo ultimo tentativo di ricerca si chiama **tabella di traduzione**. Dal momento che l'hardware non fornisce alcuna assistenza in questa fase, il sistema operativo è libero di usare il formato che preferisce. Per esempio può fare l'hashing del numero di pagina virtuale dividendolo per un numero primo p e usare il resto come indice di una tabella di puntatori, ciascuno dei quali indirizza una lista concatenata di elementi di pagine virtuali che hanno lo stesso hashing secondo p. Si noti che questi elementi non sono pagine, ma elementi di TSB. Il risultato della ricerca nella tabella di traduzione potrebbe portare a individuare la pagina in memoria, nel qual caso viene aggiornato l'elemento di TSB all'interno della cache software. Potrebbe anche succedere di scoprire che la pagina non è in memoria, per cui non resta che lanciare l'azione standard per il trattamento di un errore di pagina.

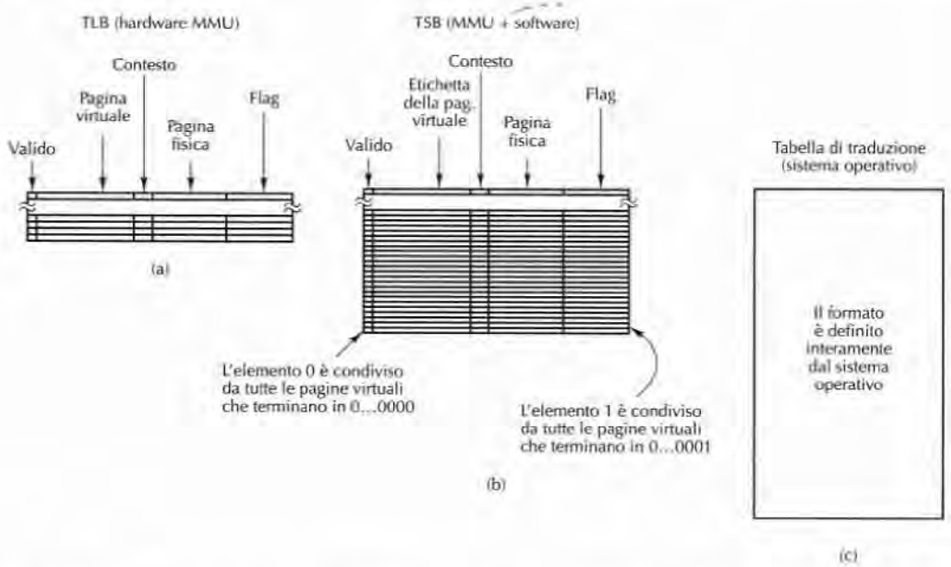


Figura 6.18 Strutture dati per la traduzione d'indirizzi virtuali dell'UltraSPARC. (a) TLB. (b) TSB. (c) Tabella di traduzione.

È interessante paragonare i due sistemi di memoria virtuale che abbiamo delineato. Il Pentium 4 supporta la segmentazione pura, la paginazione pura e i segmenti paginati. L'UltraSPARC III ha solo la paginazione. Anche il Pentium 4 usa l'hardware per percorrere la tabella delle pagine al fine di aggiornare la TLB in caso di un suo fallimento. A seguito di un fallimento della TLB, l'UltraSPARC III invece passa il controllo al sistema operativo.

La ragione principale di questa differenza è che il Pentium 4 usa segmenti di 32 bit, ed è possibile gestire segmenti così piccoli (solo un milione di pagine) per mezzo di tabelle di pagine convenzionali. In linea di principio il Pentium 4 incorrerebbe in qualche problema se un programma usasse migliaia di segmenti, ma la questione non si pone visto che nessuna versione di Windows o di UNIX consente più di un segmento per processo. La tabella convenzionale delle pagine non funzionerebbe nell'UltraSPARC III, una macchina a 64 bit che ammette 2 miliardi di pagine. In futuro tutte le macchine avranno spazi degli indirizzi virtuali di 64 bit, così lo schema dell'UltraSPARC diverrà sempre più comune.

6.1.10 Memoria virtuale e caching

La memoria virtuale con paginazione su richiesta e il caching possono sembrare a un primo sguardo completamente diversi, tuttavia sono molto simili dal punto di vista concettuale. Con la memoria virtuale il programma è mantenuto su disco e suddiviso in pagine di dimensione fissa; solo alcune di queste pagine si trovano in memoria principale. Se il programma usa prevalentemente le pagine in memoria si verificano pochi errori di pagina e il programma viene eseguito celermente. Con il caching, il programma è contenuto tutto in memoria e suddiviso in blocchi di cache di dimensione prefissata, alcuni dei quali si trovano nella cache. Se il programma usa prevalentemente i blocchi di cache, allora ci saranno pochi fallimenti di cache e l'esecuzione procederà velocemente. Dal punto di vista concettuale i due meccanismi sono la stessa cosa, ma operano a due diversi livelli gerarchici.

Naturalmente la memoria virtuale e il caching presentano anche alcune differenze. Una è che i fallimenti di cache sono gestiti dall'hardware, mentre gli errori di pagina sono gestiti dal sistema operativo. Inoltre i blocchi di cache sono in genere molto più piccoli delle pagine (per esempio 64 byte contro 8 KB). Si aggiunga che la corrispondenza tra pagine virtuali e blocchi di memoria è diversa: le tabelle delle pagine sono organizzate per essere indicizzate dai bit più significativi dell'indirizzo virtuale, mentre le cache sono indicizzate dai bit meno significativi degli indirizzi di memoria. Nonostante ciò, è importante capire che queste sono differenze di carattere implementativo, il concetto sottostante è molto simile.

6.2 Istruzioni virtuali di I/O

L'insieme d'istruzioni del livello ISA è completamente diverso dall'insieme d'istruzioni della microarchitettura. Differiscono sia le operazioni disponibili ai due livelli, sia i formati delle istruzioni. L'esistenza di poche istruzioni sostanzialmente invariate da un livello all'altro va considerata come puramente accidentale.

L'insieme d'istruzioni del livello OSM contiene invece molte delle istruzioni del livello ISA, più un numero limitato d'istruzioni importanti, meno una manciata d'istruzioni potenzialmente nocive. L'input/output è uno dei campi in cui i due livelli differiscono maggiormente e la ragione è semplice: un utente che fosse in grado di eseguire le vere istruzioni di I/O del livello ISA potrebbe leggere dati confidenziali memorizzati in qualsiasi locazione del sistema, potrebbe scrivere sui terminali degli altri utenti e, in generale, costituirebbe una minaccia per la sicurezza del sistema. In secondo luogo, nessun programmatore dotato di senno vorrebbe usare l'I/O del livello ISA, perché è estremamente noioso e complicato. Si tratta di impostare campi e bit in vari registri dei dispositivi, attendere il completamento delle operazioni e quindi verificarne l'esito. Un esempio di ciò sono i bit del registro di dispositivo dei dischi, usati in genere per rilevare gli errori seguenti (tra i tanti possibili):

1. il braccio del disco non è riuscito a portare a termine la ricerca
2. il buffer specifica locazioni di memoria inesistenti
3. l'I/O del disco è cominciato prima che venisse completata un'altra attività di I/O già iniziata
4. errore di temporizzazione della lettura
5. riferimento a un disco inesistente
6. riferimento a un cilindro inesistente
7. riferimento a un settore inesistente
8. errore di checksum del dato letto
9. errore di verifica dei dati dopo scrittura.

Quando si verifica uno di questi errori viene asserito il bit corrispondente nel registro di dispositivo. Sono pochi gli utenti che vogliono preoccuparsi di tener traccia di tutti questi bit di errore e delle altre informazioni di stato.

6.2.1 File

Un modo di organizzare l'I/O virtuale è di usare l'astrazione chiamata **file**. Nella sua forma più semplice, un file consiste in una sequenza di byte scritti su di un dispositivo di I/O. Se il dispositivo di I/O è una memoria di massa, quale un disco, il file può essere letto successivamente; se non si tratta di una memoria di massa (ma per esempio di una stampante) chiaramente il file non può più essere letto. Un disco può contenere molti file e ogni file può essere di natura diversa: un'immagine, un foglio di calcolo oppure il testo del capitolo

di un libro. File diversi hanno lunghezze e proprietà distinte. L'astrazione del file permette un'organizzazione semplice dell'I/O.

Dal punto di vista del sistema operativo un file è semplicemente una sequenza di byte. Ogni ulteriore strutturazione riguarda i programmi applicativi. L'I/O dei file è svolto tramite chiamate di sistema per l'apertura, lettura, scrittura e chiusura di file. Prima di poter essere letti i file devono essere aperti. Il procedimento di apertura di un file consente al sistema operativo di localizzarlo su disco e di caricare in memoria le informazioni necessarie per accedervi.

La chiamata di sistema per la lettura deve avere almeno i seguenti parametri:

1. un'indicazione di quale file leggere tra quelli aperti
2. un puntatore a un buffer in memoria per immagazzinare i dati letti
3. il numero di byte da leggere.

La chiamata `read` pone i dati richiesti nel buffer e in genere restituisce il numero di byte letti effettivamente (tale numero potrebbe essere inferiore al numero di byte richiesti: non è possibile leggere 2000 byte da un file di 1000 byte).

A ogni file aperto è associato un puntatore che indica il successivo byte da leggere. Dopo una lettura il puntatore è incrementato secondo il numero di byte letti, perciò letture consecutive leggono blocchi di dati consecutivi del file. In genere è possibile assegnare a questo puntatore un valore specifico, di modo da accedere direttamente a ogni locazione del file. Quando un programma ha finito di leggere un file può chiuderlo per informare il sistema operativo che non gli serve più, consentendogli così di liberare lo spazio occupato nella tabella dalle informazioni sul file.

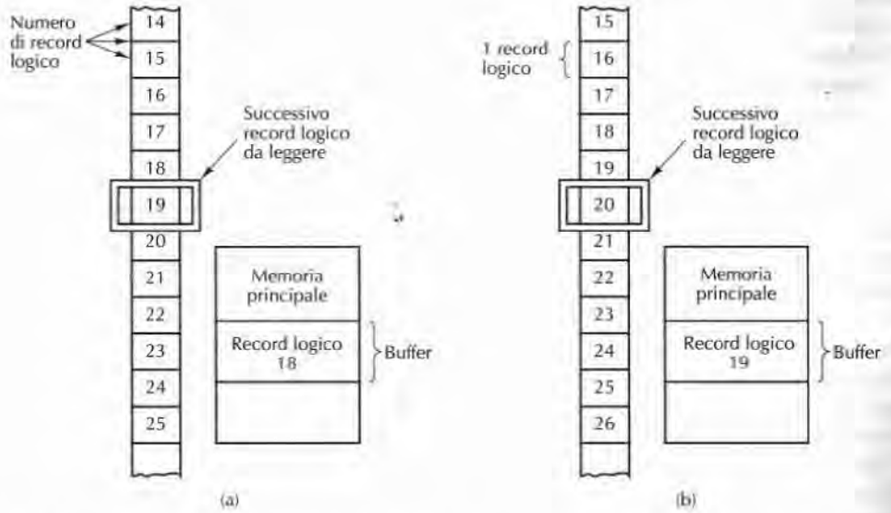


Figura 6.19 Lettura di un file costituito di record logici. (a) Prima della lettura del record 19. (b) Dopo la lettura del record 19.

I sistemi operativi dei mainframe hanno una nozione più sofisticata di file. Per loro un file è una sequenza di **record logici**, ciascuno dotato di una struttura ben definita. Un record logico, per esempio, potrebbe essere una struttura di cinque oggetti: due stringhe di caratteri, "nome" e "supervisore", due interi, "dipartimento" e "ufficio" e un valore booleano, "diSessoFemminile". Alcuni sistemi operativi fanno una distinzione tra file contenenti record con la stessa struttura e file contenenti record di tipo diverso.

L'istruzione elementare di input legge il record successivo del file specificato e lo mette in memoria principale all'indirizzo indicato, come illustrato nella Figura 6.19. Per svolgere questa operazione, l'istruzione virtuale deve sapere quale file leggere e dove riporre il record in memoria. Spesso si danno opzioni per leggere un determinato record, specificato o per posizione o tramite una chiave.

L'istruzione elementare di output scrive un record logico dalla memoria in un file. Scritture sequenziali consecutive producono nel file record logici consecutivi.

6.2.2 Implementazione delle istruzioni virtuali di I/O

Per comprendere l'implementazione delle istruzioni virtuali di I/O è necessario esaminare le modalità di organizzazione e memorizzazione dei file. Una questione basilare che devono affrontare tutti i file system (la parte del sistema operativo dedicata alla gestione dei file) è l'allocazione di memoria di massa. L'unità di allocazione può essere un singolo settore del disco, ma spesso consiste in un blocco di settori consecutivi.

Un'altra proprietà fondamentale dell'implementazione di un file system è la memorizzazione dei file in unità di allocazione consecutive o meno. La Figura 6.20 raffigura un semplice disco la cui superficie contiene 5 tracce di 12 settori ciascuna. La Figura 6.20(a) mostra uno schema di allocazione in cui i file occupano settori consecutivi, lo schema comune dei CD-ROM. La Figura 6.20(b) mostra invece uno schema in cui il settore è ancora l'unità di allocazione, ma dove un file non è vincolato a occupare settori consecutivi. Questo è lo schema di norma nei dischi.

Esiste una differenza importante tra la nozione di file dei programmatori di applicazioni e quella del sistema operativo. Il programmatore vede il file come una sequenza di byte o di record logici. Il sistema operativo vede il file come una collezione ordinata, non necessariamente consecutiva, di unità di allocazione su disco.

Per poter soddisfare la richiesta del byte o del record logico alla locazione n di un file, il sistema operativo deve sapere come localizzare i dati. Se il file è allocato in maniera consecutiva, al sistema operativo basta riconoscere l'inizio del file per calcolare la posizione del byte o del record logico richiesto.

In caso contrario, non è possibile calcolare la posizione di un byte o di un record logico arbitrario all'interno del file, a partire dalla sua sola posizione iniziale. Ci sarà dunque bisogno di una tabella contenente le unità di allocazione e i loro indirizzi su disco: la **tabella degli indici**. Quest'ultima può essere organizzata come una lista d'indirizzi di blocchi del disco (come in UNIX) o come una lista di record logici, specificandone l'indirizzo e l'offset su disco. Alle volte i record logici sono associati a una **chiave** e i programmi possono fare riferimento a un record attraverso la sua chiave, piuttosto che attraverso il suo numero. In questo caso è necessario il secondo tipo di organizzazione, che aggiunge a ogni elemento della lista la chiave del record oltre alla sua posizione su disco. Questa organizzazione è comune nei mainframe.

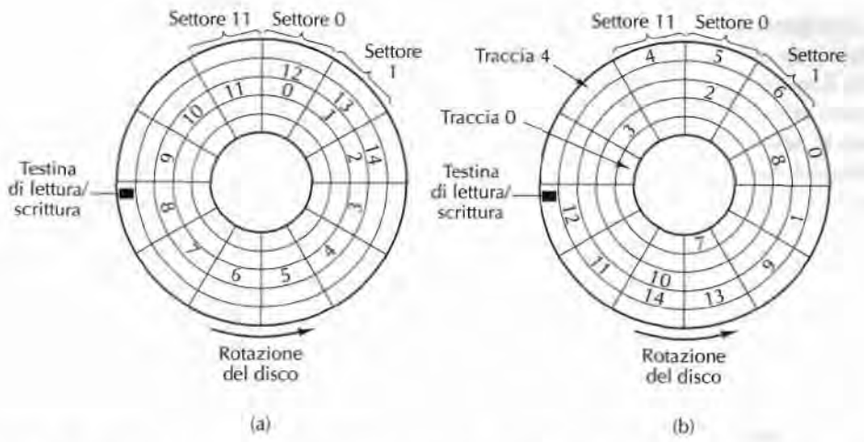


Figura 6.20 Strategie di allocazione su disco. (a) Un file in settori consecutivi. (b) Un file in settori non consecutivi.

Un altro metodo per localizzare le unità di allocazione di un file è l'organizzazione del file come una lista concatenata. Ogni unità di allocazione contiene l'indirizzo dell'unità che le succede. Per implementare in modo efficiente questo schema è necessario mantenere in memoria principale una tabella con gli indirizzi di tutte le unità successive. Per esempio, se il disco contiene 64 KB di unità d'allocazione, il sistema operativo dovrebbe gestire in memoria una tabella di 64 KB di elementi, ciascuno contenente l'indice del suo successore. Se un file occupa le unità 4, 52 e 19, l'elemento 4 della tabella dovrebbe contenere un 52, l'elemento 52 un 19, e l'elemento 19 conterrebbe un codice speciale (per esempio 0 o −1) a indicare la fine del file. Era questo il funzionamento del file system di MS-DOS e di Windows 95/98. Windows XP supporta questo schema, ma possiede anche un proprio file system nativo più simile a quello di UNIX.

Fin qui abbiamo considerato file allocati consecutivamente o meno, ma non abbiamo chiarito perché si utilizzano entrambe le soluzioni. La prima consente un'amministrazione semplice dei blocchi, ma è di difficile implementazione quando non è nota a priori la dimensione massima di file. Se un file comincia al settore j e cresce nei settori consecutivi può succedere che, se non ha abbastanza spazio per espandersi, finisca per collidere con un altro file nel settore k. Questa evenienza non crea problemi se il file non è allocato consecutivamente, poiché i blocchi successivi possono essere messi altrove sul disco. Se un disco contiene svariati file in espansione, senza che sia nota la loro dimensione finale, è pressoché impossibile memorizzarli in modo consecutivo. Il trasferimento di un file esistente è spesso possibile, ma tuttavia molto costoso.

D'altro canto, se la dimensione massima dei file è nota a priori, la situazione cambia. Per esempio il programma di masterizzazione di un CD-ROM può preallocare per ogni file una sequenza consecutiva di settori della sua lunghezza. Così i file di lunghezza 1200, 700, 200 e 900 possono essere salvati sul CD-ROM rispettivamente a partire dai settori 0, 1200, 1900 e 3900 (non prendiamo in considerazione qui la tabella dei contenuti). È facile accedere a qualsiasi locazione di un file perché il suo primo settore è noto.

Per poter allocare su disco spazio sufficiente per memorizzare i file nuovi, il sistema operativo deve tener traccia dei blocchi liberi e di quelli occupati da file. Nel caso del CD-ROM il calcolo viene effettuato all'inizio una volta per tutte, ma nel caso di un disco i file vengono salvati e cancellati di continuo. È possibile mantenere una lista di tutte le lacune, ovvero le successioni di unità di allocazione libere e contigue. Questa lista prende il nome di **lista delle locazioni libere** (*free list*). La Figura 6.21(a) illustra la lista delle locazioni libere per il disco della Figura 6.20(b).

In alternativa è possibile mantenere una bit map con un bit per ogni unità di allocazione, come mostrato nella Figura 6.21(b). I bit sono posti a 1 se l'unità di allocazione corrispondente è già occupata e azzerati in caso contrario.

La prima soluzione presenta il vantaggio di semplificare la ricerca di uno spazio libero di una data dimensione, anche se presenta lo svantaggio di richiedere una struttura dati di lunghezza variabile: la lunghezza della lista varia alla creazione o alla cancellazione di file, una caratteristica poco gradita. La bit map ha il vantaggio di avere una dimensione costante e in più richiede la sola modifica di un bit per cambiare lo stato di allocazione di un'unità, da libera a occupata o viceversa. Risulta invece difficile la ricerca di un blocco di una data lunghezza. Entrambe le tecniche richiedono l'aggiornamento della lista o della tabella all'atto di allocazione o di rimozione di un file.

Prima di terminare la trattazione dell'implementazione del file system è bene analizzare la dimensione dell'unità di allocazione, che chiama in gioco molti fattori diversi. Innanzitutto va detto che il tempo di ricerca e il ritardo rotazionale del disco predominano sui tempi di accesso. Dopo aver investito 10 ms per raggiungere l'inizio di un'unità di allocazione è decisamente preferibile leggere 8 KB (circa 1 ms) che 1 KB (0,125 ms), poiché leggere 8 KB in tranche di 1 KB richiede otto ricerche su disco. L'efficienza del trasferimento gioca a favore delle unità di allocazione grandi.

A favore di queste c'è anche il fatto che avere unità di allocazione piccole vuol dire averne molte, il che a sua volta implica indici di file lunghi o grandi tabelle di liste concatenate in memoria. Infatti una delle ragioni per cui MS-DOS è passato a unità di allocazione multisettore è dovuta al fatto che gli indirizzi del disco erano memorizzati con 16 bit. Quando i

Traccia	Settore	Numero di settori nella lacuna
0	0	5
0	6	6
1	0	10
1	11	1
2	1	1
2	3	3
2	7	5
3	0	3
3	9	3
4	3	8

(a)

Traccia	Settore 0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	1	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1	0
2	1	0	1	0	0	0	1	0	0	0	0	0
3	0	0	0	1	1	1	1	1	1	0	0	0
4	1	1	1	0	0	0	0	0	0	0	0	1

(b)

Figura 6.21 Due modi di tener traccia dei settori disponibili. (a) Con una lista delle locazioni libere. (b) Con una bit map.

dischi hanno superato i 64 KB settori, l'unico modo per rappresentarli è stato quello di usare unità di allocazione più grandi per evitare che diventassero più di 64 KB. La prima versione di Windows 95 presentava lo stesso problema, poi risolto in una versione successiva con indirizzi di 32 bit. Windows 98 supportava entrambe le dimensioni.

A favore di unità di allocazione piccole è invece la considerazione che ben pochi file occupano esattamente un numero intero di unità di allocazione. Di conseguenza quasi tutti i file sprecano dello spazio nell'ultima unità che occupano e, se un file è molto più grande dell'unità di allocazione, in media il suo spreco è dell'ordine della metà dell'unità di allocazione stessa. Più è grande l'unità di allocazione, maggiore è lo spreco: se i file sono in media molto più piccoli dell'unità di allocazione, verrà sprecato gran parte dello spazio su disco. Per fare un esempio, in una partizione di 2 GB MS-DOS o della prima versione di Windows 95, in cui le unità di allocazione erano quindi di 32 KB, un file di 100 caratteri sprecava 32.668 byte di spazio del disco. L'efficienza della memorizzazione gioca a favore delle unità di allocazione piccole. Allo stato attuale delle cose sembra prevalere l'importanza del fattore di efficienza dei trasferimenti, perciò le dimensioni dei blocchi tendono a crescere.

6.2.3 Istruzioni per la gestione di directory

Ai primordi dell'era informatica dati e programmi venivano conservati su schede perforate archiviate negli schedari dei propri uffici. Questa soluzione risultò sempre meno agevole al crescere del numero e della dimensione dei dati e dei programmi e condusse infine all'idea di usare una memoria secondaria (per esempio un disco) come alternativa agli schedari per la memorizzazione dei dati e dei programmi. Le informazioni accessibili direttamente dal computer senza bisogno dell'intervento umano si dicono **in linea** (*on-line*) in contrapposizione alle informazioni **non in linea** (*off-line*), le quali richiedono la mediazione di un operatore perché il computer possa accedervi (per esempio tramite l'inserimento di un CD-ROM).

L'informazione in linea è memorizzata nei file ed è resa accessibile ai programmi per mezzo delle istruzioni di I/O. In realtà l'informazione in linea ha bisogno d'istruzioni ulteriori per il suo mantenimento, per il raggruppamento in unità agevoli e per la protezione da tentativi di utilizzo non autorizzati.

I sistemi operativi sono soliti organizzare i file raggruppandoli in **directory** (cartelle). La Figura 6.22 raffigura un esempio di organizzazione per directory. Tutti i sistemi mettono a disposizione almeno le chiamate di sistema per assicurare le seguenti funzionalità:

1. creare un file e inserirlo in una directory
2. cancellare un file da una directory
3. rinominare un file
4. cambiare lo stato di protezione di un file.

Sono possibili diversi schemi di protezione. Uno è far sì che l'utente specifichi una password per ciascuno dei propri file: quando un programma desidera accedere al file deve fornire la password e attendere la sua convalida da parte del sistema operativo. Un altro metodo di protezione richiede a ciascun utente di specificare esplicitamente, per ogni file, una lista di utenti: solo i programmi di quegli utenti potranno accedere al file.

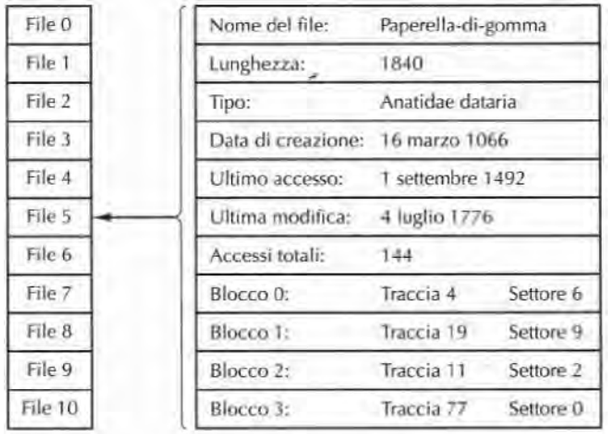


Figura 6.22 Esempio di directory di file utente e un suo tipico elemento.

Negli sistemi operativi moderni gli utenti possono gestire più di una directory. Ogni directory è a sua volta un file e può far parte così di un'altra directory, dando origine a un albero di directory. Questa molteplicità è particolarmente utile quando si lavora su più progetti allo stesso tempo: ogni directory può raggruppare tutti i file che afferiscono a un dato progetto, evitando le interferenze tra file di progetti diversi. Inoltre le directory agevolano la condivisione di file tra utenti membri di uno stesso gruppo.

6.3 Istruzioni virtuali per il calcolo parallelo

Alcuni tipi di calcolo possono essere programmati per essere svolti con maggior profitto da due o più processi cooperanti che girano in parallelo (cioè simultaneamente, su processori diversi) piuttosto che da un solo processo. Altri calcoli possono essere divisi in parti da far eseguire parallelamente, per ridurre il tempo dell'intera computazione. Nei paragrafi successivi illustreremo alcune istruzioni virtuali necessarie per consentire a più processi di collaborare in parallelo.

Le leggi della fisica forniscono un'altra ragione a favore dell'interesse attualmente rivolto al calcolo parallelo. Secondo la teoria della relatività speciale di Einstein è impossibile trasmettere un segnale elettrico a una velocità superiore a quella della luce, che è di circa 30 cm/ns (300.000 km/s) nel vuoto, e un po' inferiore nei fili di rame e nelle fibre ottiche. Questo limite ha implicazioni importanti sulla strutturazione dei computer. Per esempio se la CPU richiede dati alla memoria principale e questa si trova a 30 cm, ci vorrà almeno 1 ns perché la richiesta giunga alla memoria e ancora 1 ns perché la risposta giunga alla CPU. Perciò i computer che vorranno lavorare in tempi sotto il nanosecondo dovranno essere molto miniaturizzati o, in alternativa, essere dotati di molte CPU. Un computer con mille CPU da 1 ns può raggiungere la stessa potenza di calcolo di una CPU con cicli di 0,001 ns, ed è molto più facile e più economico da costruire.

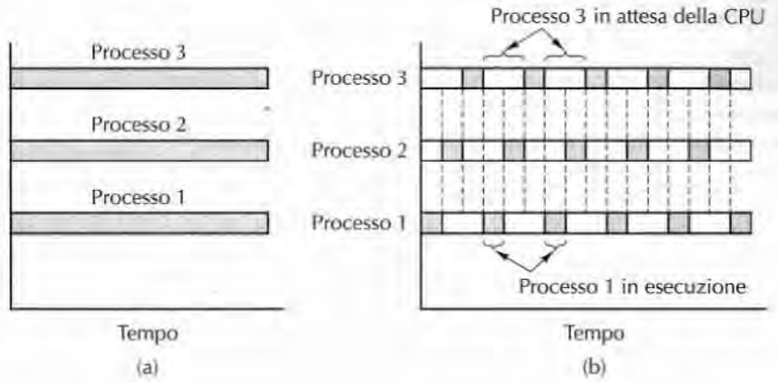


Figura 6.23 (a) Calcolo genuinamente parallelo su CPU multiple. (b) Calcolo parallelo simulato tramite l'alternanza della singola CPU all'interno di un insieme di tre processi.

Su un calcolatore con più di una CPU è possibile assegnare ciascun processo cooperativo a una CPU, facendo sì che progrediscano tutti simultaneamente. Se invece c'è una sola CPU è possibile simulare l'effetto dell'elaborazione parallela alternando l'esecuzione dei diversi processi per brevi lassi di tempo. In altre parole, il processore è condiviso dai diversi processi.

La Figura 6.23 mostra la differenza tra il vero calcolo parallelo, possibile con più processori, e quello simulato, attuato su di un solo processore fisico. Anche quando il parallelismo è solo simulato risulta utile guardare a ogni processo come se disponesse di un processore virtuale dedicato. Nella simulazione si verificano gli stessi problemi di comunicazione riscontrabili nel vero calcolo parallelo.

6.3.1 Creazione dei processi

Ogni programma in esecuzione fa parte di un processo. Un processo è caratterizzato da uno stato e da uno spazio degli indirizzi tramite cui accedere ai dati e al programma. Lo stato comprende almeno il program counter, una `PSW`, un puntatore allo stack e dei registri d'uso generale.

Quasi tutti i sistemi operativi moderni consentono la creazione e la terminazione dinamica dei processi. Esiste perciò una chiamata di sistema per la creazione di un processo, fondamentale per realizzare il calcolo parallelo. Questa chiamata di sistema può limitarsi a clonare il processo chiamante oppure gli può consentire di specificare stato iniziale, programma, dati e indirizzo iniziale del nuovo processo.

In alcuni casi il processo creatore (genitore) mantiene controllo parziale o anche totale del processo creato (figlio). A tal fine esistono istruzioni virtuali perché il processo genitore possa fermare, far ripartire, esaminare e porre fine al processo figlio. In altri casi il genitore ha meno controllo sul figlio: una volta creato il processo figlio, il genitore non ha modo di forzarlo a fermarsi, ripartire, farsi esaminare o terminare la propria esecuzione. I due processi procedono nelle rispettive attività in modo indipendente l'uno dall'altro.

6.3.2 Corsa critica

In molti casi i processi paralleli hanno bisogno di comunicare e di sincronizzarsi al fine di poter svolgere il proprio compito. Qui ci avvaliamo di un esempio dettagliato per trattare la sincronizzazione tra processi ed esaminare alcune delle difficoltà che comporta; nel paragrafo successivo forniremo le soluzioni a queste problematiche.

Si consideri la situazione di due processi indipendenti, il processo 1 e il processo 2, che comunicano attraverso un buffer condiviso in memoria principale; per comodità chiamiamo il primo **produttore** e il secondo **consumatore**. Il produttore calcola i numeri primi e li ripone nel buffer uno per volta. Il consumatore li rimuove dal buffer uno alla volta e li visualizza sullo schermo.

Questi due processi girano in parallelo con velocità diverse. Se il produttore si rende conto che il buffer è pieno allora *va a dormire*, ovvero sospende temporaneamente le sue operazioni in attesa di un segnale dal consumatore. Successivamente il consumatore, dopo aver rimosso un numero dal buffer, invia il segnale al produttore per *svegliarlo*, cioè per farlo ripartire. Analogamente, il consumatore va a dormire quando il buffer è vuoto. Il produttore lo sveglierà non appena avrà inserito un numero nel buffer vuoto.

In questo esempio usiamo un buffer circolare per la comunicazione tra processi. I puntatori *in* e *out* sono usati nel modo seguente: *in* punta alla parola libera successiva (dove il produttore inserirà il prossimo numero primo), *out* punta al numero che sarà rimosso per primo dal consumatore. Il buffer è vuoto quando *in* = *out*, come nella Figura 6.24(a). La Figura 6.24(b) mostra la situazione dopo che il produttore ha generato alcuni numeri primi. La Figura 6.24(c) mostra la situazione dopo che il consumatore ha rimosso e stampato alcuni di quei numeri primi. Le Figure 6.24(d)-(f) illustrano una possibile evoluzione del buffer a seguito dell'attività dei due processi. La cima del buffer è logicamente contigua con la sua base, ossia il buffer si riavvolge in modo circolare. Se si verifica una raffica di inserimenti tali che *in* punti alla parola precedente di quella puntata da *out* (per esempio *in* = 52 e *out* = 53) allora il buffer è pieno. L'ultima parola libera non è utilizzata, perché altrimenti non ci sarebbe alcun modo per capire se *in* = *out* indica il buffer vuoto o il buffer pieno.

La Figura 6.25 mostra una semplice implementazione in Java del problema produttore-consumatore. Questa soluzione usa tre classi: `m`, `producer` e `consumer`. La classe principale `m` contiene alcune definizioni di costanti, i puntatori al buffer *in* e *out* e il buffer stesso, che in questo esempio può contenere 100 primi da `buffer[0]` a `buffer[99]`.

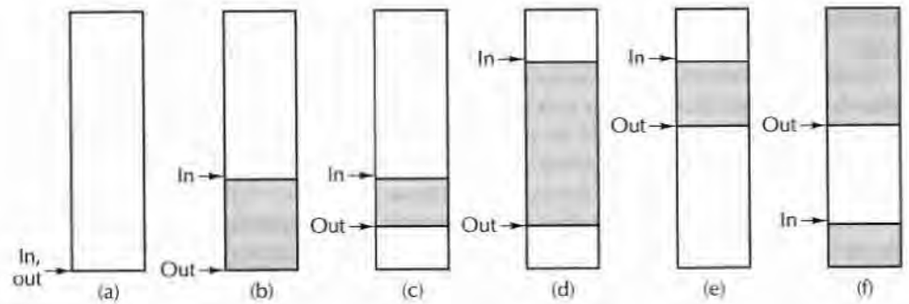


Figura 6.24 Uso di un buffer circolare.


```
public class m {
   final public static int BUF_SIZE = 100;                    // il buffer va da 0 a 99
   final public static long MAX_PRIME = 1000000000000L;       // valore d'arresto
   public static int in = 0, out = 0;                         // puntatori ai dati
   public static long buffer[ ] = new long[BUF_SIZE];         // memorizza i numeri primi
   public static producer p;                                  // nome del produttore
   public static consumer c;                                  // nome del consumatore

   public static void main(String args[ ]) {                  // classe main
      p = new producer( );                                    // crea il produttore
      c = new consumer( );                                    // crea il consumatore
      p.start( );                                             // avvia il produttore
      c.start( );                                             // avvia il consumatore
   }
   // funzione di servizio per l'incremento circolare di in e out
   public static int next(int k) {if (k < BUF_SIZE - 1) return(k+1); else return(0);}
}

class producer extends Thread {                               // classe produttore
   public void run( ) {                                       // codice del produttore
      long prime = 2;                                         // variabile di lavoro

      while (prime < m.MAX_PRIME) {
         prime = next_prime(prime);                           // istruzione P1
         if (m.next(m.in) == m.out) suspend( );               // istruzione P2
         m.buffer [m.in] = prime;                             // istruzione P3
         m.in = m.next(m.in);                                 // istruzione P4
         if (m.next(m.out) == m.in) m.c.resume( );            // istruzione P5
      }
   }

   private long next_prime(long prime){ ... }                 // funzione che calcola il numero primo successivo

}


class consumer extends Thread {                               // classe consumatore
   public void run( ) {                                       // codice consumatore
      long emirp = 2;                                         // variabile di lavoro

      while (emirp < m.MAX_PRIME) {
         if (m.in == m.out) suspend( );                       // istruzione C1
         emirp = m.buffer [m.out];                            // istruzione C2
         m.out = m.next(m.out);                               // istruzione C3
         if (m.out == m.next(m.next(m.in))) m.p.resume( );    // istruzione C4
         System.out.println(emirp);                           // istruzione C5
      }
   }
}
```

Figura 6.25 Calcolo parallelo in situazione fatale di corsa critica.

Il codice si avvale dei thread Java per simulare il calcolo parallelo. Le classi `producer` e `consumer` sono istanziate rispettivamente nelle variabili `p` e `c`; sono entrambe classi derivate dalla classe base `Thread` provvista del metodo `run`, che serve a contenere il codice

del thread. L'invocazione del metodo `start` di un oggetto derivato da `Thread` provoca l'avvio di un nuovo thread (l'esecuzione del suo metodo `run`).

Un thread somiglia a un processo, con la differenza che tutti i thread di un programma Java accedono allo stesso spazio degli indirizzi; questa caratteristica permette loro di condividere un buffer. Se un calcolatore ha due o più CPU è possibile assegnare ciascun thread a una diversa CPU e ottenere quindi il vero parallelismo. Se c'è una sola CPU, i thread la condividono secondo lo schema di condivisione del tempo. Continueremo a far riferimento ai processi produttore e consumatore (visto che il nostro interesse è per i processi paralleli) anche se Java supporta solo i thread paralleli e non i processi veramente paralleli.

La funzione `next` serve a incrementare *in* e *out* in modo agevole, senza dover controllare ogni volta il verificarsi di un riavvolgimento circolare. Se il parametro di `next` è minore o uguale a 98 la funzione restituisce l'intero successivo. Se invece il parametro vale 99 abbiamo raggiunto la fine del buffer e la funzione restituisce 0.

C'è bisogno poi di un modo per consentire ai processi di dormire quando non possono continuare l'esecuzione. I progettisti di Java hanno previsto questa necessità e hanno incluso i metodi `suspend` (sonno) e `resume` (risveglio) nella classe `Thread` sin dalla prima versione di Java. Se ne fa uso nella Figura 6.25.

Veniamo quindi al codice del produttore e del consumatore vero e proprio. All'inizio il produttore genera un nuovo numero primo nell'istruzione P1. Si noti qui l'uso di `m.MAX_PRIME`: il prefisso `m.` indica che intendiamo il valore `MAX_PRIME` definito nella classe `m`. Per le stesse ragioni `m.` è anche il prefisso di `in`, `out`, `buffer` e `next`.

In seguito il produttore controlla (in P2) se *in* è arretrato di una posizione rispetto a *out*. In tal caso (per esempio *in* = 62, *out* = 63) il buffer è pieno e il produttore va a dormire invocando `suspend` in P2. Se il buffer non è pieno il nuovo primo può essere inserito nel buffer (P3) e *in* incrementato (P4). Se in P5 il nuovo valore di *in* eccede di 1 *out* (per esempio *in* = 17 e *out* = 16), dovevano essere necessariamente uguali prima dell'incremento di *in*. Il produttore conclude che il buffer era precedentemente vuoto e che il consumatore sta ancora dormendo, dunque chiama la `resume` per svegliarlo (P5). Infine il produttore comincia a preoccuparsi del successivo valore da produrre.

Il programma del consumatore è strutturato in modo simile. All'inizio effettua un test (C1) per verificare se il buffer è vuoto; in caso affermativo non c'è lavoro da svolgere e il consumatore può andare a dormire. In caso negativo, il consumatore rimuove il successivo numero da stampare (C2) e incrementa *out* (C3). A questo punto, se *out* è due posizioni oltre *in* (C4) vuol dire che precedentemente distavano l'un l'altro una sola posizione; poiché *in* = *out* − 1 è la condizione di "buffer pieno", il produttore deve essere ancora in fase di sonno; quindi il consumatore lo risveglia chiamando la `resume`. Infine stampa il numero (C5) e ricomincia lo stesso ciclo.

Sfortunatamente questa soluzione contiene un difetto fatale, illustrato nella Figura 6.26. Si ricordi che i due processi girano in modo asincrono ed, eventualmente, a diverse velocità. Si consideri il caso della Figura 6.26(a), dove il buffer contiene un solo elemento nel posto 21, con *in* = 22 e *out* = 21. Se il produttore si trova all'istruzione P1 in cerca di un numero primo e il consumatore è occupato in C5 a stampare l'elemento di posizione 20, allora il consumatore termina la stampa del numero, effettua il test di C1 e in C2, preleva dal buffer l'ultimo numero presente, quindi incrementa *out*. In questo istante *in* e *out* valgono entrambi 22. Poi il consumatore stampa il numero e torna a C1, dove legge *in* e *out* dalla memoria per effettuarne il confronto, come mostrato nella Figura 6.26(b).

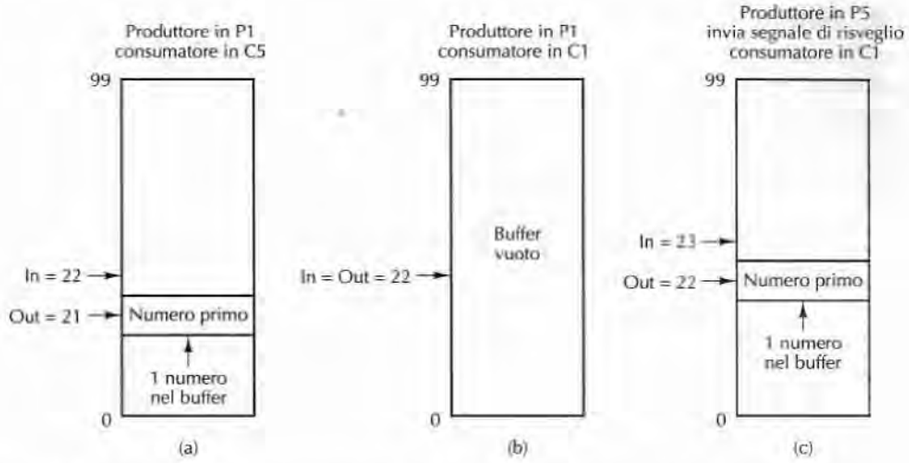


Figura 6.26 Insuccesso del meccanismo di comunicazione tra produttore e consumatore.

In questo preciso istante, dopo che il consumatore ha prelevato *in* e *out* dalla memoria e prima di averli confrontati, il produttore trova il numero primo successivo, lo inserisce nel buffer con l'istruzione P3 e incrementa *in* in P4. In altre parole, *in* vale ora *out* + 1 a indicare la presenza di un elemento nel buffer e il produttore ne deduce (erroneamente) che il consumatore è sospeso, perciò gli invia un segnale di risveglio (cioè chiama `resume`), come raffigurato dalla Figura 6.26(c). Invece il consumatore è ancora sveglio e così il segnale di risveglio va perso. Il produttore inizia la ricerca di un altro numero primo.

Adesso il consumatore continua la propria attività: ha già recuperato *in* e *out* dalla memoria prima dell'inserimento nel buffer dell'ultimo numero trovato dal produttore. Poiché entrambi i puntatori valgono 22 il consumatore va a dormire. Successivamente il produttore trova un nuovo numero primo, osserva che *in* = 24 e *out* = 22, e conclude che ci sono due numeri nel buffer (vero) e che il consumatore è sveglio (falso). Il produttore prosegue la sua esecuzione e arriva il momento in cui riempie il buffer e va a dormire. Ora entrambi i processi sono dormienti e lo rimarranno per sempre.

Il problema qui è stato originato dal fatto che, nel lasso di tempo tra la lettura di *in* e *out* da parte del consumatore e il suo assopimento, il produttore è intervenuto di soppiatto, ha scoperto che *in* = *out* + 1, ha pensato che il consumatore stesse dormendo (non ancora vero) e gli ha spedito un segnale di sveglia che è andato perso proprio perché il consumatore era ancora sveglio. Questa problematica è nota con il nome di **corsa critica**, perché il successo del metodo dipende dal vincitore della corsa al test di *in* e *out* dopo l'incremento di *out*.

Il problema delle corse critiche è ben noto ed è così serio che, svariati anni dopo l'introduzione di Java, Sun ha cambiato l'interfaccia della classe `Thread` rinnegando l'uso di `suspend` e `resume` perché conducevano spesso a condizioni di corsa critica. La soluzione alternativa proposta si basa sul linguaggio, ma visto che ci interessiamo qui ai sistemi operativi, prendiamo in considerazione un'ulteriore soluzione fornita da molti di loro, compresi UNIX e XP.

6.3.3 Sincronizzazione dei processi tramite semafori

È possibile risolvere le situazioni di corsa critica in almeno due modi. Una soluzione consiste nel dotare ciascun processo di un "bit risveglio pendente". Ogni volta che viene spedito un segnale di risveglio a un processo ancora attivo il suo bit risveglio pendente viene asserito. Se un processo va a dormire con questo bit asserito, la sua esecuzione riprende immediatamente e il bit viene azzerato. Il bit risveglio pendente serve a conservare un segnale di risveglio superfluo per un uso futuro.

Anche se questo metodo risolve la corsa critica tra due processi, fallisce però nel caso generale di comunicazione tra n processi, perché bisognerebbe salvare $n - 1$ bit di risvegli pendenti per contare fino a $n - 1$, ma si tratta evidentemente di una soluzione astrusa.

Dijkstra (1968b) ha proposto una soluzione più generale al problema della sincronizzazione tra processi paralleli che prevede l'uso di variabili intere (non negative) chiamate **semafori**. Il sistema operativo fornisce due chiamate di sistema, `up` e `down`, per agire sui semafori: la prima per incrementare un semaforo di 1, la seconda per decrementarlo di 1.

Se si effettua un'operazione `down` di un semaforo che ha valore positivo, questo viene decrementato di 1 e il processo corrispondente continua. Invece se il semaforo vale 0 la `down` non può andare a buon fine; il processo corrispondente è messo a dormire e resta in questo stato finché un altro processo esegue una `up` su quel semaforo. In genere i processi dormienti sono posti in una coda al fine di poterne tener traccia.

L'istruzione `up` su un semaforo controlla se il semaforo è a 0, nel qual caso lo incrementa di un'unità e consente al processo dormiente di completare l'operazione `down` che ha causato la sua sospensione; il risultato di queste operazioni è che il semaforo vale ancora 0, ma entrambi i processi possono ora continuare la loro esecuzione. Un'istruzione `up` su di un semaforo non nullo, lo incrementa di 1. Un semaforo serve essenzialmente a contare i segnali di risveglio per uso futuro, così che non vadano persi. Una proprietà fondamentale delle istruzioni sui semafori è che, una volta che un processo ha intrapreso un'operazione su di un semaforo, nessun altro può accedervi finché il primo non ha completato l'istruzione o finché non resta sospeso nel tentativo di effettuare una `down` su 0. La Figura 6.27 riassume le proprietà essenziali delle chiamate di sistema `up` e `down`.

Come già menzionato, Java possiede una soluzione basata sul linguaggio per trattare le corse critiche: occupandoci qui di sistemi operativi, cerchiamo un modo di rappresentare in Java i semafori anche se non fanno parte del linguaggio o delle classi standard. Lo facciamo ipotizzando l'esistenza di due metodi nativi, `up` e `down`, che svolgono i compiti delle

Istruzione	Semaforo = 0	Semaforo > 0
Up	Semaforo = semaforo + 1; se un altro processo era stato arrestato nel tentativo di completare un'istruzione down su questo semaforo, adesso può completare la down e proseguire la propria esecuzione	Semaforo = semaforo + 1
Down	Il processo resta sospeso finché un altro processo non esegue una up su questo semaforo	Semaforo = semaforo – 1

Figura 6.27 Risultato delle operazioni su semafori.


```
public class m {
   final public static int BUF_SIZE = 100;                    // il buffer va da 0 a 99
   final public static long MAX_PRIME = 1000000000000L;       // valore d'arresto
   public static int in = 0,out = 0;                          // puntatori ai dati
   public static long buffer[ ] = new long[BUF_SIZE];         // memorizza i numeri primi
   public static producer p;                                  // nome del produttore
   public static consumer c;                                  // nome del consumatore
   public static int filled = 0, available = 100;             // semafori

   public static void main(String args[ ]) {                  // classe main
      p = new producer( );                                    // crea il produttore
      c = new consumer( );                                    // crea il consumatore
      p.start( );                                             // avvia il produttore
      c.start( );                                             // avvia il consumatore
   }
   // funzione di servizio per l'aggiornamento circolare di in e out
   public static int next(int k) {if (k < BUF_SIZE – 1) return(k+1); else return(0);}
}

class producer extends Thread {                               // classe produttore
   native void up(int s); native void down(int s);            // metodi sui semafori
   public void run( ) {                                       // codice del produttore
      long prime = 2;                                         // variabile di lavoro

      while (prime < m.MAX_PRIME) {
         prime = next_prime(prime);                           // istruzione P1
         down(m.available);                                   // istruzione P2
         m.buffer[m.in] = prime;                              // istruzione P3
         m.in = m.next(m.in);                                 // istruzione P4
         up(m.filled);                                        // istruzione P5
      }
   }

   private long next_prime(long prime){ ... }                 // funzione che calcola il numero primo successivo
}

class consumer extends Thread {                               // classe consumatore
   native void up(int s); native void down(int s);            // metodi sui semafori
   public void run( ) {                                       // codice del consumatore
      long emirp = 2;                                         // variabile di lavoro

      while (emirp < m.MAX_PRIME) {
         down(m.filled);                                      // istruzione C1
         emirp = m.buffer[m.out];                             // istruzione C2
         m.out = m.next(m.out);                               // istruzione C3
         up(m.available);                                     // istruzione C4
         System.out.println(emirp);                           // istruzione C5
      }
   }
}
```

Figura 6.28 Calcolo parallelo con i semafori.

chiamate di sistema `up` e `down` (rispettivamente) e che accettano in ingresso parametri interi. Questi metodi ci consentono di esprimere l'uso dei semafori in Java.

La Figura 6.28 mostra come eliminare la corsa critica tramite l'uso dei semafori. Aggiungiamo alla classe `m` i due semafori `available` (disponibile), che inizialmente vale 100, e `filled` (riempito), posto inizialmente a 0. Come prima, il produttore comincia l'esecuzione da P1 e il consumatore parte da C1. L'esecuzione di `down` su `filled` interrompe

immediatamente il processo consumatore. Quando il produttore ha trovato il numero primo invoca `down` con parametro `available`, assegnandogli il valore 99. All'istruzione P5 chiama `up` con parametro `filled`, assegnandogli il valore 1 e liberando così il consumatore dall'attesa; questi può ora completare la sua chiamata `down` sospesa. In questo istante `filled` vale 0 ed entrambi i processi sono in esecuzione.

Torniamo alla precedente situazione di corsa critica in una situazione in cui *in* = 22, *out* = 21, il produttore si trova in P1 e il consumatore in C5. Dunque il consumatore completa l'istruzione e torna a C1, dove invoca `down` su `filled`, che passa così da 1 a 0. Poi il consumatore preleva l'ultimo numero disponibile nel buffer, chiama `up` su `available`, che arriva a 100, stampa a schermo il valore e torna nuovamente in C1. Un attimo prima che il consumatore chiami `down`, il produttore trova il successivo numero primo ed esegue in rapida successione le istruzioni P2, P3 e P4.

Adesso `filled` vale 0. Il produttore è sul punto di eseguire una `up` su `filled` e il consumatore una `down` sullo stesso parametro. Se il consumatore esegue per primo l'istruzione rimarrà sospeso finché il produttore non lo libererà (chiamando la `up`). Se invece parte per primo il produttore, questi porrà il semaforo a 1 e il consumatore non verrà sospeso. In entrambi i casi non viene perso alcun segnale di risveglio; era proprio l'obiettivo che ci eravamo proposti introducendo i semafori.

La proprietà essenziale delle operazioni sui semafori è che sono indivisibili: una volta iniziata un'operazione su di un semaforo, nessun altro processo può usare il semaforo finché il primo non abbia completato l'operazione o non rimanga sospeso. Per di più, l'uso dei semafori scongiura la perdita dei segnali di risveglio. L'istruzione `if` della Figura 6.25 non è indivisibile: è possibile che un processo spedisca un segnale di risveglio tra la valutazione della condizione e l'esecuzione dell'istruzione selezionata.

Il problema della sincronizzazione di processi è stato risolto dichiarando indivisibili le chiamate di sistema `up` e `down`, richiamate dai metodi `up` e `down`. Perché ciò accada, il sistema operativo deve proibire che un semaforo venga usato da più di un processo per volta, o può almeno limitarsi a interrompere l'esecuzione di ogni altro processo finché quello impiegato nella chiamata di sistema non la porti a compimento. Sui sistemi monoprocessore i semafori sono implementati, qualche volta, mediante la disattivazione degli interrupt durante le operazioni sui semafori. Sui sistemi multiprocessore questo stratagemma non funziona.

La sincronizzazione mediante semafori funziona per un numero arbitrario di processi. Ci possono essere allo stesso tempo molti processi dormienti, impegnati nel tentativo di completare una chiamata di sistema `down` sullo stesso semaforo. Quando un altro processo effettua una `up` su quel semaforo, uno dei processi in attesa può completare finalmente la `down` e riprendere l'esecuzione. Il valore del semaforo resta 0 e gli altri processi restano in attesa.

Usiamo un'analogia per chiarire la natura dei semafori. Immaginate un torneo con 20 squadre di pallavolo suddiviso in 10 partite (i processi), ciascuna disputata sul proprio campo, e una cassa (il semaforo) contenente i palloni. Sfortunatamente ci sono solo 7 palloni disponibili (il semaforo assume valori tra 0 e 7). L'inserimento di un pallone nella cassa equivale a una `up` perché incrementa il valore del semaforo; l'estrazione di un pallone corrisponde a una `down` poiché ne decrementa il valore.

All'inizio del torneo ogni campo invia un giocatore al canestro per appropriarsi di un pallone. Sette di loro riusciranno nell'impresa (completano la `down`), tre sono costretti ad aspettare un pallone (cioè non riescono a completare la `down`). Le loro partite sono temporaneamente sospese; prima o poi una delle altre partite finirà e il rispettivo pallone verrà riportato nel canestro (esecuzione di una `up`). Grazie a questa operazione uno dei tre giocatori in attesa potrà accaparrarsi la palla (completando la sua `down` incompiuta) e dar via alla propria partita. Gli altri due giocatori restano in attesa che vengano riportati nel cesto altri palloni; solo allora le ultime due partite potranno essere disputate.

6.4 Sistemi operativi di esempio

In questo paragrafo procediamo con la trattazione dei nostri sistemi d'esempio, il Pentium 4 e l'UltraSPARC III. Per ciascuno analizzeremo un sistema operativo: per il Pentium 4 ci riferiamo a Windows XP (d'ora in poi solo XP), e per l'UltraSPARC III a UNIX. Cominciamo da quest'ultimo perché è più semplice e per molti versi più elegante. Inoltre questo ordine è più sensato e naturale dal momento che UNIX è stato progettato e implementato molto tempo prima di XP e lo ha assai influenzato.

6.4.1 Introduzione

UNIX

UNIX venne sviluppato presso i laboratori Bell nei primi anni '70. La prima versione, scritta da Ken Thompson in linguaggio assemblativo del minicomputer PDP-7, venne presto seguita da una versione per il PDP-11 (scritta nell'allora nuovo linguaggio C, ideato e sviluppato da Dennis Ritchie). Nel 1974 Ritchie e Thompson pubblicarono un articolo su UNIX che è rimasto una pietra miliare del settore (Ritchie e Thompson, 1974). In ragione del lavoro descritto in quell'articolo furono successivamente insigniti del prestigioso ACM Turing Award (Ritchie, 1984; Thompson, 1984). La pubblicazione dell'articolo stimolò molte università a richiedere una copia di UNIX ai laboratori Bell. Essendo questi di proprietà della AT&T, ai tempi un'azienda monopolista cui non era concesso fare affari nel mondo dell'informatica, la richiesta venne accolta senza difficoltà e la licenza di UNIX venne ceduta alle università per una modica cifra.

Per una di quelle coincidenze che spesso fanno la storia, il PDP-11 era il computer prediletto dai dipartimenti di informatica di quasi tutte le università e i sistemi operativi di cui era equipaggiato erano considerati terribili tanto dai professori quanto dagli studenti. UNIX colmò velocemente questa lacuna, specialmente perché era distribuito con tutti i codici sorgenti e così chiunque poteva armeggiarci liberamente, cosa che fecero in molti.

Una delle prime università a procurarsi UNIX fu la University of California, a Berkeley. Avendo a disposizione tutto il codice sorgente, i ricercatori di Berkeley poterono modificare il sistema in modo sostanziale. Tra i risultati più importanti ci fu l'implementazione sul minicomputer VAX, l'aggiunta della memoria virtuale paginata, l'estensione dei nomi di file da 14 a 255 caratteri e l'inclusione del protocollo di rete TCP/IP, usato ancor oggi in Internet (soprattutto perché si trovava nello UNIX di Berkeley all'avvento delle reti).

Mentre Berkeley procedeva nelle sue modifiche, la stessa AT&T continuava a sviluppare UNIX, giungendo al System III nel 1982 e al System V nel 1984. Alla fine degli anni '80 circolavano due versioni di UNIX diverse e abbastanza incompatibili: UNIX Berkeley e

System V. Questa scissione del mondo di UNIX, unita al fatto che mancavano gli standard per i formati dei programmi binari, inibì fortemente il successo commerciale di UNIX perché nessun fornitore di software avrebbe potuto scrivere e confezionare programmi UNIX che funzionassero su ogni sistema UNIX (a differenza di MS-DOS). Dopo molti litigi, il comitato per gli standard IEEE creò uno standard chiamato **POSIX** (*Portable Operating System-IX*, "sistema operativo portabile-IX"), noto anche con il nome di P1003 (il suo numero di standard IEEE). Successivamente POSIX è divenuto uno standard internazionale.

Lo standard è diviso in molte sezioni, ciascuna dedicata a una parte differente di UNIX. La prima sezione, P1003.1, definisce le chiamate di sistema; la seconda sezione, P1003.2, definisce le funzionalità basilari e così via; P1003.1 definisce circa 60 chiamate di sistema che devono essere fornite da tutti i sistemi conformi allo standard. Si tratta delle chiamate elementari per la lettura e scrittura di file, per la creazione di nuovi processi, e così via. Praticamente tutti i sistemi UNIX oggi in commercio supportano le chiamate di sistema P1003.1 e in genere molti ne supportano anche altre, in special modo quelle definite da System V e/o da UNIX Berkeley. Queste chiamate di sistema addizionali possono raggiungere facilmente le 200 unità.

Nel 1987 l'autore di questo libro distribuì il codice sorgente di MINIX, una piccola versione di UNIX, per uso accademico (Tanenbaum, 1987). Linus Torvald, dopo aver studiato MINIX presso la sua università di Helsinki e dopo averlo installato sul suo PC di casa, decise di scrivere un proprio clone di MINIX, battezzato Linux e divenuto da allora molto popolare. MINIX e Linux sono entrambi conformi allo standard e tutto quanto verrà detto in questo capitolo su UNIX si applica anche a loro, se non specificato diversamente. Malgrado queste varianti di UNIX differiscano molto al loro interno, in questo capitolo parleremo per lo più dell'interfaccia delle loro chiamate di sistema, che accomuna tutte le versioni.

Il sistema operativo dell'UltraSPARC III si chiama **Solaris** e si basa sul System V, ma supporta allo stesso tempo gran parte delle chiamate di sistema di Berkeley.

La Figura 6.29 fornisce una prima suddivisione in categorie delle chiamate di sistema di Solaris. Le categorie più numerose e importanti comprendono le chiamate di sistema per la gestione dei file e delle directory, molte delle quali appartengono a P1003.1. Buona parte delle altre chiamate di sistema deriva invece dal System V.

Categoria	Alcuni esempi
Gestione dei file	Apertura, lettura, scrittura, chiusura e lock di file
Gestione delle directory	Creazione e cancellazione delle directory; trasferimento di file
Gestione dei processi	Generazione, terminazione, tracing dei processi e invio di segnali tra loro
Gestione della memoria	Condivisione di memoria tra processi; protezione di pagine
Lettura e scrittura di parametri	Consultazione dell'ID di un utente, di un gruppo, di un processo; assegnamento della priorità
Data e ora	Impostazione dell'ora di accesso a file; timer d'intervallo; profilo esecutivo
Rete	Imposta/accetta connessioni; invia/riceve messaggi
Miscellanea	Creazione di utenti; manipolazione delle quote su disco; riavvio del sistema

Figura 6.29 Suddivisione approssimativa delle chiamate di sistema UNIX.

La categoria delle chiamate di rete è invece dovuta soprattutto a UNIX Berkeley, che ha introdotto il concetto di **socket**, ossia il punto terminale di una connessione di rete, modellata sull'esempio delle prese a muro per le connessioni telefoniche. Un processo UNIX può creare una socket, collegarsi a essa e stabilire una connessione a una socket di un'altra macchina, grazie alla quale può scambiare dati in modo bidirezionale, avvalendosi in genere del protocollo TCP/IP. Una frazione rilevante dei server di Internet funzionano con UNIX proprio perché la sua tecnologia di rete esiste da decenni ed è ormai stabile e matura.

È difficile fare delle asserzioni generali sulla struttura del sistema operativo UNIX perché ne esistono molte implementazioni, ciascuna a suo modo diversa dalle altre. Ciò nondimeno, la Figura 6.30 vale per quasi tutte le implementazioni. Alla sua base si trova il livello dei driver di dispositivo che mette il sistema operativo al riparo dal contatto con l'hardware vero e proprio. In un primo momento i driver di dispositivo venivano scritti come entità indipendenti e separate dagli altri driver, causando un'inutile reiterazione degli sforzi, visto che molti driver devono affrontare problematiche analoghe, quali il controllo del flusso, la gestione degli errori, le priorità, la separazione dei dati dal controllo, e così via. Questa osservazione suggerì a Dennis Ritchie lo sviluppo del modello **stream** ("flusso") per la scrittura di driver modulari. Uno stream permette di stabilire una connessione bidirezionale tra un processo dell'utente e un dispositivo hardware e di inserire uno o più moduli lungo il percorso. Il processo dell'utente invia dati lungo lo stream e questi vengono elaborati o trasformati da ciascun modulo finché non giungono all'hardware. I dati provenienti dai dispositivi subiscono l'elaborazione inversa.

Al di sopra dei driver di dispositivo c'è il file system che gestisce i nomi dei file, le directory, l'allocazione dei blocchi del disco, la protezione e altro ancora. Parte del file system è la **cache a blocchi** che memorizza i blocchi letti dal disco più di recente in caso dovessero tornare utili a breve. Nel corso degli anni sono stati utilizzati molti file system diversi, tra cui il fast file system di Berkeley (McKusick et al., 1984) e file system a struttura logaritmica (Rosenblum e Ousterhout, 1991; Seltzer et al., 1993).

Un'altra parte del kernel UNIX riguarda la gestione dei processi che, tra le altre cose, si occupa della IPC (*InterProcess Communication*, "comunicazione tra processi"). La IPC

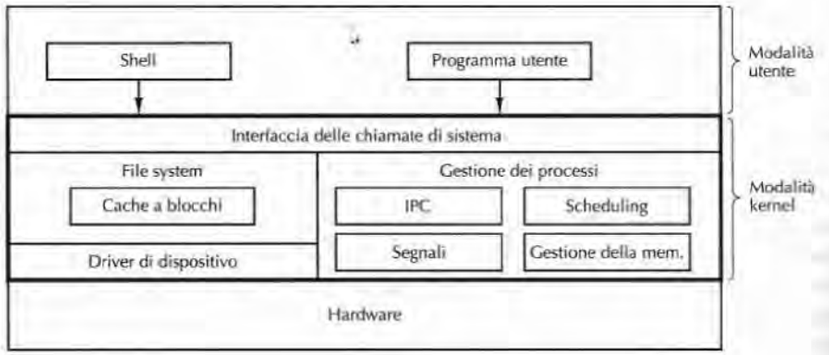


Figura 6.30 Struttura di un comune sistema UNIX.

mette a disposizione svariati meccanismi che consentono ai processi di comunicare tra loro e di sincronizzarsi per evitare le corse critiche. Il codice per la gestione dei processi gestisce anche lo scheduling dei processi in base alle loro priorità. Anche i segnali, che costituiscono una forma di interrupt software (asincrono), sono gestiti a questo livello, così come la memoria. La gran parte dei sistemi UNIX supporta la memoria virtuale con paginazione su richiesta e alle volte fornisce anche delle caratteristiche in più, come la possibilità di condividere regioni dello spazio degli indirizzi tra più processi.

Sin dalla sua nascita, UNIX fu pensato come un sistema dalle dimensioni contenute al fine di incrementarne affidabilità e prestazioni. La prima versione di UNIX era tutta testuale e usava terminali che potevano visualizzare 24 o 25 righe di 80 caratteri ASCII. L'interfaccia utente era gestita dalla **shell** ("conchiglia, involucro"), un programma che girava a livello utente e offriva un'interfaccia a linea di comando. Poiché la shell non faceva parte del kernel era facile aggiungere a UNIX nuove shell, e con il passare del tempo ne sono state realizzate molte altre, sempre più sofisticate.

In seguito all'introduzione dei primi terminali grafici, venne sviluppato presso il M.I.T. un sistema a finestre per UNIX chiamato **X Windows**, poi raffinato mediante l'aggiunta di **Motif**, una **GUI** (*Graphical User Interface*, "interfaccia grafica per l'utente") più elaborata. Quasi tutto il codice di X Windows e di Motif gira a livello utente, fuori dal kernel, perché si attiene alla filosofia di UNIX che vuole il kernel di dimensioni contenute.

Windows XP

Al suo lancio nel 1981, il primo PC IBM era equipaggiato con MS-DOS 1.0, un sistema operativo a 16 bit, a singolo utente e orientato alla linea di comando. Questo sistema operativo era costituito da 8 KB di codice residente in memoria. Due anni dopo fece la sua comparsa MS-DOS 2.0, un sistema molto più potente da 24 KB; conteneva un elaboratore di linea di comando (la shell) più un certo numero di caratteristiche prese in prestito da UNIX. Nel 1984 IBM lanciò sul mercato i PC/AT basati sul 286 ed equipaggiati con MS-DOS 3.0, che aveva ormai raggiunto i 36 KB. Con il passare degli anni MS-DOS continuò ad acquisire nuove funzionalità, ma restava pur sempre un sistema orientato alla linea di comando.

Ispirata dal successo dei Macintosh di Apple, Microsoft decise di dotare MS-DOS di un'interfaccia grafica per l'utente che chiamò **Windows**. Le prime tre versioni di Windows, culminate nella versione 3.x, non erano veri sistemi operativi, bensì interfacce grafiche per l'utente montate su MS-DOS, che manteneva ancora il controllo della macchina. Tutti i programmi giravano nello stesso spazio degli indirizzi e così un baco in uno di loro poteva arrestare malamente l'intero sistema.

La distribuzione di Windows 95 nel 1995 mantenne MS-DOS, anche se introdusse la nuova versione 7.0. Windows 95 e MS-DOS 7.0 assommavano insieme quasi tutte le caratteristiche di un sistema operativo completo, compresa la memoria virtuale, la gestione dei processi e la multiprogrammazione. Tuttavia Windows 95 non era un programma a 32 bit effettivi, ma conteneva grosse porzioni di codice a 16 bit (affiancate al codice a 32 bit) e si appoggiava ancora sul file system di MS-DOS, ereditandone quasi tutti i limiti. I soli cambiamenti di una certa importanza furono l'aggiunta del supporto dei nomi di file lunghi, a estendere gli 8 + 3 caratteri consentiti in MS-DOS, e la possibilità di avere più di 65.536 blocchi su un disco.

MS-DOS sopravvisse anche al lancio di Windows 98 nel 1998 (anche se era arrivato nel frattempo alla versione 7.1) e continuò a eseguire codice a 16 bit. Nonostante la migrazione di qualche altra funzionalità dalla parte MS-DOS verso Windows e l'organizzazione dei dischi che rendeva standard il supporto dei dischi grandi, al di là delle apparenze, Windows 98 non era poi così diverso da Window 95. La differenza principale stava nell'interfaccia per l'utente che integrava in modo più stretto il desktop, Internet e la riproduzione video. Fu questa integrazione ad attirare l'attenzione del Dipartimento di Giustizia degli Stati Uniti che citò Microsoft in giudizio per aver assunto una posizione illegale di monopolio. A Windows 98 succedette per breve tempo Windows Millennium Edition (ME) che ne costituiva una versione leggermente migliorata.

Parallelamente a questi sviluppi, Microsoft stava lavorando alacremente al completamento di un nuovo sistema operativo a 32 bit, per la cui scrittura era partita da zero, e che chiamò **Windows New Technology** o **Windows NT**. In un primo momento venne presentato come il sistema operativo che avrebbe rimpiazzato tutti gli altri nei PC basati su Intel, ma per qualche ragione ebbe una diffusione lenta e venne più tardi ricollocato nelle fasce alte di mercato, dove conquistò una propria nicchia. La seconda versione di NT prese il nome di Windows 2000 e divenne la versione di punta, anche nel mercato dei desktop. XP è il successore di Windows 2000, ma si tratta in questo caso di cambiamenti secondari, per cui XP è essenzialmente una versione migliorata di Windows 2000.

XP è commercializzato in due versioni, server e client, pressoché identiche e generate a partire dallo stesso codice sorgente. La versione server è stata ideata per le macchine usate come server di file e di stampa su LAN e comprende perciò alcune funzionalità gestionali più elaborate di quelle della versione client, destinata invece ai desktop con singolo utente. La versione server è distribuita in alcune varianti (enterprise) ideate per grandi installazioni. Le varianti vengono messe a punto diversamente, ottimizzandole per l'ambiente in cui ci si aspetta di installarle. Al di là di queste differenze, le versioni sono tutte molto simili e infatti quasi tutti i file eseguibili sono identici in tutte le versioni. Lo stesso XP, per stabilire la propria versione, accede a una variabile conservata in una struttura dati interna (il registro) cui è interdetto l'accesso agli utenti, perché non possano modificarla e convertire la versione client (economica) nelle versioni server o enterprise (molto più costose). Non faremo più alcuna distinzione tra le varie versioni di XP.

Mentre MS-DOS e tutte le versioni precedenti di Windows erano sistemi a utente singolo, XP supporta la multiprogrammazione, consentendo a più utenti per volta di lavorare sulla stessa macchina. Per esempio un server di rete potrebbe avere più utenti attivi allo stesso tempo sulla rete, consentendo a ciascuno l'accesso ai propri file.

XP è un sistema operativo per la multiprogrammazione, a 32 bit: supporta la coesistenza di più processi utente, ognuno dotato di un proprio spazio degli indirizzi virtuali di 32 bit e paginato a richiesta. XP stesso è scritto tutto in codice di 32 bit.

Uno dei miglioramenti innovativi di NT rispetto a Windows 95 era la sua struttura modulare basata su un kernel relativamente piccolo, la cui esecuzione avveniva in modalità kernel, più svariati processi server in esecuzione in modalità utente. I processi utente interagivano con i processi server sul modello client-server: un client invia un messaggio di richiesta al server, il server svolge il lavoro e restituisce il risultato al client con un secondo messaggio. La struttura modulare consentì una facile portabilità su molti altri calcolatori non appartenenti alla linea Intel, compresi i DEC Alpha, i PowerPC IBM e i MIPS SGI. C'è da

dire però che, per ragioni di prestazioni, a partire da NT 4.0 gran parte del sistema è stato reinserito nel kernel.

Potremmo certo dilungarci molto sulla struttura di XP e sull'interfaccia delle sue chiamate di sistema, ma il nostro interesse principale verte sull'aspetto delle macchine virtuali indotte dai diversi sistemi operativi (cioè dalle loro chiamate di sistema), perciò diamo qui un breve resoconto della struttura del sistema per passare poi all'interfaccia delle chiamate di sistema.

La Figura 6.31 illustra la struttura di XP, costituita da un certo numero di moduli organizzati a livelli e che contribuiscono a implementare il sistema operativo. A ogni modulo corrisponde una funzione particolare e un'interfaccia ben definita con gli altri moduli. Quasi tutti i moduli sono scritti in C, anche se una parte dell'interfaccia del dispositivo grafico è scritta in C++ e una piccola porzione dei livelli più bassi è scritta in linguaggio assemblativo.

Alla base della struttura c'è un sottile **livello di astrazione hardware** (*hardware abstraction layer*) che ha lo scopo di presentare al resto del sistema operativo dispositivi hardware astratti e privi degli spigoli e delle idiosincrasie di cui abbonda l'hardware reale. Tra i dispositivi modellati ci sono le cache non residenti nel chip, i timer, i bus di I/O, i controllori di interrupt e i DMA. La loro presentazione in forma idealizzata al resto del sistema operativo facilita la portabilità di XP su altre piattaforme hardware, dal momento che tutte le modifiche da effettuare sono concentrate.

Al di sopra del livello di astrazione hardware si trova un livello che contiene il kernel e i driver di dispositivo; entrambi possono accedere direttamente all'hardware visto che contengono porzioni di codice che dipendono dall'hardware stesso.

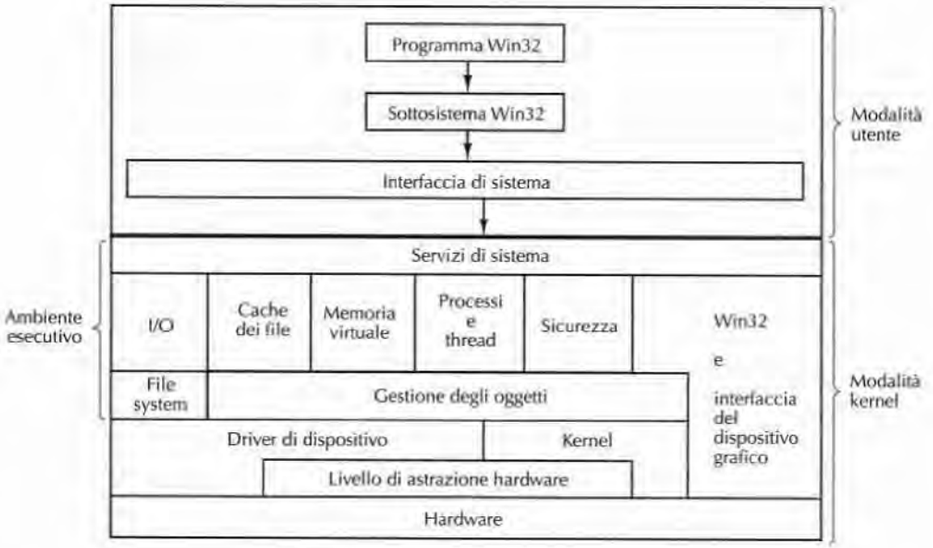


Figura 6.31 Struttura di Windows XP.

Il **kernel** supporta gli oggetti primitivi del kernel, cioè gli interrupt e le trap, oltre alla gestione delle eccezioni, lo scheduling dei processi, la sincronizzazione multiprocesso e la temporizzazione. L'obiettivo di questo livello è di rendere la parte restante del sistema operativo completamente indipendente dall'hardware e perciò facilmente portabile. Il kernel risiede permanentemente in memoria principale ed è *non preemptible*, anche se può cedere il controllo temporaneamente per l'assistenza agli interrupt di I/O.

Ogni **driver di dispositivo** può controllare uno o più dispositivi di I/O, ma può anche svolgere delle azioni che non sono legate a un dispositivo specifico, come cifrare flussi di dati o anche soltanto fornire accesso alle strutture dati del kernel. Dato che gli utenti possono installare nuovi driver di dispositivo, essi hanno il potere di intaccare il kernel e pregiudicare lo stato del sistema; per questa ragione è importante che i driver vengano scritti con molta cura.

Al di sopra del kernel e dei driver dei dispositivi si trova la parte superiore del sistema operativo, detta **esecutiva** (*executive*). Questa parte (suddivisa a sua volta in tre livelli) dipende dall'architettura e può essere portata su macchine differenti con uno sforzo relativamente modesto.

Il livello inferiore contiene il file system e il gestore degli oggetti. Il **file system** gestisce l'uso dei file e delle directory, e il **gestore degli oggetti** (*object manager*) gestisce gli oggetti conosciuti dal kernel, tra cui i processi, i thread (processi leggeri contenuti in un solo spazio degli indirizzi), i file, le directory, i semafori, i dispositivi di I/O, i timer e altro. Il gestore degli oggetti controlla anche uno spazio dei nomi in cui vengono inseriti gli oggetti di nuova creazione affinché siano successivamente reperibili.

Il livello successivo consta di sei parti principali, come illustrato nella Figura 6.31. Il **gestore di I/O** mette a disposizione uno schema generale per la gestione di dispositivi di I/O e fornisce alcuni servizi generici di I/O.

Il **gestore della cache dei file** è addetto alla gestione dei blocchi dei file e assiste il gestore della memoria virtuale nella scelta dei blocchi da mantenere in memoria, ed è responsabile della gestione dei file mappati in memoria. XP può essere configurato con file system multipli, nel qual caso il gestore della cache svolge il proprio lavoro per conto di tutti. Quando si rende necessario reperire un blocco, questo viene richiesto al gestore della cache; se non è presente in cache, il gestore si rivolge al file system appropriato per ottenerlo. Il gestore della cache interagisce con il gestore della memoria virtuale perché i file possono essere mappati negli spazi degli indirizzi dei processi e si richiede quindi un controllo di coerenza dei dati.

Il **gestore della memoria virtuale** implementa l'architettura di memoria virtuale paginata a richiesta di XP. Gestisce la corrispondenza tra pagine virtuali e blocchi di memoria, e assicura la validità delle regole di protezione secondo cui ogni processo può accedere alle sole pagine appartenenti al proprio spazio degli indirizzi (eccetto in determinate circostanze speciali). Gestisce inoltre alcune chiamate di sistema che coinvolgono la memoria virtuale.

Il **gestore dei processi e dei thread** si occupa dei meccanismi di funzionamento di processi e thread per ciò che riguarda la loro creazione e distruzione, ma non le loro politiche di utilizzo.

Il **monitor di sicurezza** (*security reference monitor*) rafforza il già elaborato meccanismo di sicurezza di XP che soddisfa i requisiti del Libro arancione C2 del Dipartimento della Difesa degli Stati Uniti. Il Libro arancione detta un gran numero di regole cui si devono

conformare i sistemi informatici, a partire dall'autenticazione tramite login, passando per la gestione del controllo degli accessi, per finire al fatto che le pagine virtuali devono essere azzerate prima di essere riutilizzate.

L'**interfaccia del dispositivo grafico** si occupa della gestione delle immagini per lo schermo e la stampante. Fornisce agli utenti le chiamate di sistema per scrivere sul monitor o sulla stampante in modo indipendente dal dispositivo. Contiene inoltre i driver del dispositivo hardware per l'uscita grafica; nelle versioni precedenti a XP 4.0 i driver si trovavano nello spazio utente, ma le prestazioni erano molto deludenti. Microsoft li ha poi trasferiti nel kernel per renderli più veloci. Anche il modulo Win32, che gestisce gran parte delle chiamate di sistema, si trovava originariamente nello spazio utente ed è stato successivamente portato nel kernel.

In cima alla parte esecutiva c'è un livello sottile di **servizi di sistema** che fornisce un'interfaccia alla parte esecutiva: riceve le chiamate XP vere e proprie e le smista agli altri componenti della parte esecutiva.

All'esterno del kernel si trovano i programmi utente e il **sottosistema ambientale** (*environmental subsystem*). Quest'ultimo è stato previsto per scoraggiare i programmi utente dall'effettuare direttamente le chiamate di sistema (anche se è tecnicamente possibile) e mette loro a disposizione un insieme di chiamate di funzioni. Originariamente esistevano tre sottosistemi ambientali: Win32 (per i programmi di NT, Windows 2000, XP o anche di Windows 95/98/ME), POSIX (per i programmi UNIX) e OS/2 (per i programmi OS/2). Dei tre solo Win32 è supportato, anche se esiste un nuovo modulo di servizi che garantisce un modesto supporto di UNIX.

Le applicazioni di Windows usano le funzioni di Win32 e comunicano con il sottosistema Win32 per effettuare le chiamate di sistema. Il **sottosistema Win32** accetta le chiamate di funzioni Win32 (descritte in seguito) e usa il modulo di libreria **interfaccia di sistema** (che è in realtà un file DLL, si veda al riguardo il Capitolo 7) per effettuare le vere chiamate di sistema di XP, necessarie a svolgere il compito richiesto.

Dopo aver esaminato brevemente la struttura di XP, torniamo ora al nostro interesse principale: i servizi offerti da XP. La loro interfaccia è il punto di contatto tra il programmatore e il sistema. Purtroppo Microsoft non ha mai reso pubblica la lista completa delle chiamate di sistema XP, che tuttavia cambia da distribuzione a distribuzione. Tali carenze rendono praticamente impossibile la scrittura di programmi che possano effettuare chiamate di sistema dirette.

Microsoft ha preferito rendere pubblico un insieme di chiamate detto **API di Win32** (*Application Programming Interface*, "interfaccia per la programmazione delle applicazioni"). Si tratta di procedure di libreria che alle volte si avvalgono delle chiamate di sistema per eseguire il compito loro richiesto, alle altre operano direttamente nella procedura di libreria all'interno dello spazio utente o nel sottosistema Win32. Le chiamate API di Win32 non cambiano con le versioni di Windows, in modo da garantire la stabilità delle applicazioni. Ciononostante, esistono anche alcune chiamate API di XP che possono però variare tra le diverse versioni. Ci limitiamo qui allo studio delle API di Win32 perché, anche se non esauriscono tutte le chiamate del sistema, sono ben documentate e più stabili nel tempo. In seguito all'adattamento di Windows alle macchine a 64 bit, Microsoft ha cambiato il nome di Win32 per includere la versione a 64 bit, ma ai nostri fini basterà l'analisi della prima versione.

La filosofia delle API di Win32 è completamente diversa da quella di UNIX. Nel caso di UNIX le chiamate di sistema sono tutte pubbliche e formano un'interfaccia minimale: l'eliminazione di una sola di loro ridurrebbe la funzionalità del sistema operativo. Al contrario Win32 fornisce un'interfaccia ridondante (ci sono spesso tre o quattro modi diversi di fare la stessa cosa) che comprende anche alcune funzioni che non dovrebbero essere (e che infatti non sono) chiamate di sistema, per esempio una chiamata API per la copia di un file intero.

Molte chiamate API di Win32 servono a creare oggetti del kernel, come file, processi, thread, pipe, e così via, e come risultato restituiscono al chiamante l'**handle** ("impugnatura") dell'oggetto. L'handle può poi essere usato per svolgere le operazioni definite sull'oggetto. L'handle appartiene al processo che ha creato l'oggetto corrispondente e non può essere passato direttamente a un altro processo perché ne faccia uso (nello stesso modo in cui i descrittori di file UNIX non possono essere passati ad altri processi perché ne facciano uso). Tuttavia, in determinate circostanze è possibile duplicare un handle e passarne la copia a un altro processo in modo protetto, consentendogli l'accesso a un oggetto che non gli appartiene. Un handle può essere accompagnato da un descrittore di sicurezza che specifica in dettaglio quali processi possono operare su di esso, e con quali operazioni.

Qualche volta si sente parlare di XP come di un sistema orientato agli oggetti, perché l'unico modo di manipolare gli oggetti del kernel è di invocare metodi (funzioni API) sui loro handle. D'altra parte, mancano a XP alcune delle proprietà fondamentali dei sistemi orientati agli oggetti quali l'ereditarietà e il polimorfismo.

Le API di Win32 erano disponibili anche in Windows 95/98/ME (e su Windows CE, il sistema operativo per l'elettronica di consumo), a parte qualche eccezione. Per esempio Windows 95/98 erano sprovvisti della sicurezza, perciò le chiamate API che hanno a che fare con la sicurezza restituirebbero su Windows 95/98 una segnalazione d'errore. Inoltre, i nomi di file di XP usano l'insieme di caratteri Unicode, che non era ancora disponibile su Windows 95/98. Ci sono differenze anche nel tipo dei parametri di alcune chiamate di funzioni API. Per esempio, in XP le coordinate dello schermo che si trovano nelle funzioni grafiche sono numeri a 32 bit effettivi; in Windows 95/98 venivano usati solo i 16 bit meno significativi (per la retrocompatibilità con Windows 3.1). Se da una parte, l'esistenza delle API di Win32 nei diversi sistemi operativi facilita la portabilità dei programmi, dall'altra evidenzia ancor di più la distanza che esiste tra di loro e le chiamate di sistema vere e proprie.

6.4.2 Esempi di memoria virtuale

In questo paragrafo affrontiamo la memoria virtuale di UNIX e di XP che, dal punto di vista del programmatore, sono molto simili.

Memoria virtuale di UNIX

Tutti i processi UNIX hanno tre segmenti: codice, dati e stack (Figura 6.32). Nelle macchine dotate di un solo spazio lineare d'indirizzi, in genere si pone il codice alla base della memoria, lo si fa seguire dai dati e si alloca lo stack in cima alla memoria stessa. La dimensione del codice è fissa, mentre i dati e lo stack sono liberi di crescere in direzioni opposte. Questo modello è molto semplice da implementare su qualsiasi macchina ed è quello usato da Solaris.

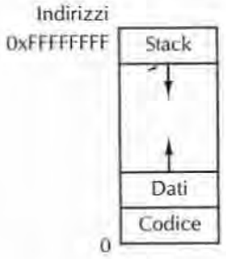


Figura 6.32 Spazio degli indirizzi di un processo UNIX.

Inoltre, se la macchina consente la paginazione, l'intero spazio degli indirizzi può essere paginato senza che il programmatore neanche se ne accorga. L'unica cosa che il programmatore sa è che può prendersi più spazio per i suoi programmi di quanto ce ne sia in memoria fisica. Per consentire comunque la condivisione di tempo tra un numero arbitrario di processi, i sistemi UNIX che non dispongono della paginazione, in genere fanno lo swap tra memoria e disco di interi processi per volta.

Per quanto riguarda la memoria virtuale (con paginazione a richiesta) dello UNIX Berkeley non resta da aggiungere altro. Per il System V (e quindi Solaris) invece va aggiunto che esso comprende diverse funzionalità che consentono agli utenti una gestione molto più sofisticata della loro memoria virtuale. Di particolare importanza è la possibilità che un processo mappi un file (o una sua parte) nel proprio spazio degli indirizzi. Per esempio, se un file di 12 KB è mappato all'indirizzo virtuale 144 KB, la lettura della parola all'indirizzo 144 KB equivale alla lettura della prima parola del file. Grazie a questo meccanismo è possibile leggere e scrivere file senza ricorrere alle chiamate di sistema. Dato che la dimensione di un file può eccedere quella dello spazio degli indirizzi virtuali, è possibile mapparne in memoria anche solo una parte. L'operazione comincia con l'apertura del file che restituisce il suo descrittore *fd*, usato successivamente per identificare il file corrispondente da mappare. Quindi il processo effettua la chiamata

```
paddr = mmap(indirizzo_virtuale, lunghezza, protezione, flag, fd,
        file_offset)
```

che mappa `lunghezza` byte, a partire dalla posizione `offset` all'interno del file, negli indirizzi virtuali che cominciano a `indirizzo_virtuale`. In alternativa è possibile impostare il parametro `flag` in modo che sia il sistema a scegliere un indirizzo virtuale, restituito nella variabile `paddr`. La regione mappata deve comprendere un numero intero di pagine e deve essere allineata alle loro estremità. Il parametro `protezione` può specificare una combinazione qualsiasi di permessi in lettura, scrittura o esecuzione. La corrispondenza può essere poi rimossa con la chiamata `unmap`.

Più processi possono mappare lo stesso file contemporaneamente. La condivisione può avvenire secondo due opzioni. La prima prevede che tutte le pagine siano condivise, perciò le scritture effettuate da un processo sono visibili a tutti gli altri. Questa opzione mette a disposizione un percorso per la comunicazione tra processi con larghezza di banda elevata. L'altra opzione prevede la condivisione delle pagine fintanto che non siano state modificate; non appena un processo tenta di scrivere in una pagina si verifica un errore di protezione, a seguito

del quale il sistema operativo fornisce al processo una copia privata della pagina su cui scrivere. Questo schema, noto con il nome di **copia dopo scrittura** (*copy on write*), torna utile quando si vuol dare a ogni processo l'illusione di essere l'unico a mappare un certo file.

Memoria virtuale di Windows XP

In XP ogni processo utente ha il proprio spazio degli indirizzi virtuali. Gli indirizzi virtuali sono lunghi 32 bit, così ogni processo ha uno spazio degli indirizzi virtuali di 4 GB. I primi 2 GB sono disponibili per il codice e i dati, i successivi 2 GB permettono l'accesso (limitato) alla memoria del kernel, fatta eccezione per le versioni Server di Windows, in cui la suddivisione è di 3 GB per l'utente e 1 GB per il kernel. Lo spazio degli indirizzi virtuali è paginato a richiesta, con una dimensione di pagina fissa (4 KB nel Pentium 4).

Le pagine virtuali si possono trovare in uno dei tre stati seguenti: libero, riservato o impegnato. Una **pagina libera** è una pagina non attualmente in uso; un riferimento a essa causa un errore di pagina. All'avvio di un processo tutte le sue pagine sono libere finché non comincia la corrispondenza tra programma e dati nel suo spazio degli indirizzi. Una volta che una parte del codice o dei dati è mappata in una pagina, questa si dice **impegnata** (*committed*). Un riferimento a una pagina impegnata viene mappato per mezzo dell'hardware della memoria virtuale e ha successo solo se la pagina si trova in memoria principale. In caso contrario, si verifica un errore di pagina e il sistema operativo si preoccupa di caricare la pagina dal disco. Una pagina virtuale può anche essere **riservata** per indicare che non è disponibile alla corrispondenza, a meno che la riserva venga rimossa esplicitamente. Oltre a questi attributi, una pagina può anche essere leggibile, scrivibile o eseguibile. I primi e gli ultimi 64 KB di memoria sono sempre liberi al fine di intercettare gli errori di puntamento (i puntatori non inizializzati valgono in genere 0 o −1).

A ciascuna pagina impegnata corrisponde su disco una pagina ombra (*shadow page*) dove viene conservata quando non è presente in memoria principale. Le pagine libere o riservate non hanno una pagina ombra, perciò i riferimenti a esse causano errori di pagina (il sistema non può caricare una pagina dal disco se questa non esiste). Le pagine ombra sono organizzate su disco in uno o più file di paginazione. Il sistema operativo si preoccupa di tener traccia della corrispondenza tra le pagine virtuali e le parti dei file di paginazione. Nel caso del testo del programma (con permesso di sola esecuzione) è il file binario eseguibile a contenere le pagine ombra; le pagine di dati sono invece contenute in file speciali.

XP, al pari di System V, permette la corrispondenza diretta tra file e regioni dello spazio degli indirizzi virtuali (cioè in sequenze di pagine consecutive). Una volta mappato nello spazio degli indirizzi, un file può essere letto o scritto tramite le comuni operazioni di riferimento alla memoria.

I file mappati in memoria sono implementati allo stesso modo delle pagine impegnate, con la sola differenza che le pagine ombra si trovano in questo caso nei file stessi invece che nel file di paginazione. Di conseguenza, è possibile che la versione di un file mappato in memoria sia diversa da quella su disco (a causa di scritture recenti nello spazio degli indirizzi virtuali). In ogni caso, il file viene aggiornato quando estromesso dalla memoria o a seguito di una richiesta esplicita di *flush* (che forza una scrittura dei cambiamenti su disco).

XP permette espressamente la corrispondenza contemporanea di un file da parte di due o più processi, anche a indirizzi virtuali differenti. Grazie alla lettura e scrittura di parole di memoria, i processi possono comunicare tra loro e scambiarsi dati con una larghezza di

banda molto elevata, poiché non è richiesta alcuna operazione di copia. Processi diversi possono avere permessi d'accesso diversi. Poiché tutti i processi che mappano lo stesso file condividono le stesse pagine, i cambiamenti effettuati da uno di loro sono visibili immediatamente a tutti gli altri, anche se non è ancora stata aggiornata la copia su disco.

Le API di Win32 contengono diverse funzioni per consentire a un processo la gestione esplicita della propria memoria virtuale; le più importanti sono elencate nella Figura 6.33. Tutte queste funzioni operano su regioni costituite da una singola pagina o da una sequenza di due o più pagine consecutive nello spazio degli indirizzi virtuali.

Le prime quattro funzioni API sono autoesplicative. Con le altre due, un processo può costringere la paginazione a tenere un certo numero di pagine in memoria o può rimuovere questo vincolo. La cosa può essere utile alle applicazioni in tempo reale, ma è consentita solo ai programmi dell'amministratore di sistema. Inoltre, il sistema operativo stabilisce dei vincoli cui devono attenersi i processi perché non diventino troppo avidi. Seppur non elencate nella Figura 6.33, XP dispone di funzioni API per consentire a un processo di accedere alla memoria virtuale di un altro processo per il quale ha ricevuto il controllo (cioè di cui ha un handle).

Le ultime quattro funzioni API elencate servono alla gestione dei file mappati in memoria. La mappatura di un file comincia con la creazione di un oggetto di mappatura file mediante `CreateFileMapping`. La funzione restituisce un handle dell'oggetto di mappatura file e, se richiesto, lo associa a un nome nel file system perché possa essere usato da un altro processo. Le due funzioni seguenti servono rispettivamente alla corrispondenza e alla rimozione di file dalla memoria. L'ultima funzione può essere invocata da un processo per mappare un file già mappato da un altro processo. In questo modo, due o più processi possono condividere regioni dei rispettivi spazi degli indirizzi.

Queste sono le funzioni API basilari attorno alle quali è costruito il sistema di gestione della memoria. Per fare un esempio, esistono funzioni API per l'allocazione o la rimozione di strutture dati in uno o più *heap* ("mucchio, cumulo"). Gli heap sono usati per la memorizzazione di quelle strutture dati che vengono create e distrutte dinamicamente e che

Funzione API	Significato
VirtualAlloc	Riserva o impegna una pagina
VirtualFree	Libera o disimpegna una pagina
VirtualProtect	Cambia la protezione in lettura/scrittura/esecuzione di una regione
VirtualQuery	Esamina lo stato di una regione
VirtualLock	Rende una regione residente in memoria (cioè ne disabilita la paginazione)
VirtualUnlock	Rende una regione paginabile nel modo consueto
CreateFileMapping	Crea un oggetto di mappatura file e gli assegna un nome (se specificato)
MapViewOfFile	Colloca (una parte di) un file nello spazio degli indirizzi
UnmapViewOfFile	Rimuove un file mappato dallo spazio degli indirizzi
OpenFileMapping	Apre un oggetto di mappatura file precedentemente creato

Figura 6.33 Le più importanti chiamate API di Windows XP per la gestione della memoria virtuale.

non sono oggetto di *garbage collection* (l'operazione di rimozione automatica delle strutture dati inutilizzate a cura del sistema), perciò sta al software dell'utente liberare i blocchi di memoria virtuale che non sono più in uso. La gestione dello heap in XP è simile alla funzione `malloc` dei sistemi UNIX, con la differenza che in XP ci possono essere più heap gestiti indipendentemente.

6.4.3 Esempi di I/O virtuale

L'obiettivo principale del sistema operativo è quello di fornire servizi ai programmi utente, soprattutto servizi di I/O come la scrittura e la lettura di file. Sia UNIX sia XP offrono un ampio ventaglio di servizi di I/O ai programmi utente. Se è vero che esiste una chiamata XP equivalente a ogni chiamata UNIX, non vale l'inverso, perché XP possiede molte più chiamate e ogni sua chiamata è molto più complicata della controparte UNIX.

I/O virtuale di UNIX

Buona parte del successo di UNIX risiede nella sua semplicità che, a sua volta, è una conseguenza diretta dell'organizzazione del file system. Un file ordinario è una sequenza lineare di byte che comincia alla posizione 0 e prosegue fino a un massimo di $2^{32} - 1$ byte. Il sistema operativo non impone alcuna struttura ai file, anche se gli utenti interpretano i file di testo ASCII come una successione di righe, ciascuna completata da un carattere di fine linea.

A ogni file aperto è associato un puntatore al byte successivo da leggere o in cui scrivere. Le chiamate di sistema `read` e `write` leggono o scrivono dati a partire dalla posizione nel file indicata dal puntatore. Entrambe le chiamate implicano l'avanzamento del puntatore di un numero di posizioni uguale al numero di byte trasferiti. È tuttavia possibile l'accesso diretto a una posizione del file fornendone il valore al puntatore.

Oltre ai file ordinari, UNIX supporta anche file speciali, usati per accedere ai dispositivi di I/O. In genere ogni dispositivo di I/O è associato a uno o più file speciali. Un programma può utilizzare il dispositivo di I/O attraverso la lettura o scrittura su questi file. È questo il modo in cui vengono gestiti dischi, stampanti, terminali e molti altri dispositivi.

La Figura 6.34 elenca le chiamate di UNIX più importanti relative al file system. La chiamata `creat` (CREATE sarebbe sbagliato!) è utilizzabile per creare un nuovo file, anche se non è più necessaria perché nelle ultime versioni anche la funzione `open` è in grado di farlo. `unlink` cancella un file, nell'ipotesi che questo si trovi in una sola directory.

La chiamata `open` si usa per aprire file esistenti (e per crearne di nuovi). L'indicatore `mode` specifica la modalità d'apertura (in lettura, scrittura, e così via). La chiamata restituisce come risultato il **descrittore di file**, un piccolo intero che serve a identificare il file nelle chiamate successive.

L'I/O propriamente detto si svolge con le chiamate `read` e `write`, le quali prendono in ingresso il descrittore del file da usare, un buffer per i dati da leggere o contenente i dati da scrivere, più un valore `count` che stabilisce la quantità di dati da trasmettere. `lseek` serve a spostare il puntatore al file, in modo da realizzare l'accesso diretto alle sue posizioni.

La chiamata `stat` serve per ottenere informazioni sui file, quali la dimensione, la data dell'ultimo accesso, il proprietario e altro ancora. `chmod` modifica la modalità di protezione del file, per esempio consentendo o negando la lettura del file agli utenti che non

Chiamata di sistema	Significato
creat(name, mode)	Crea un file (*mode* specifica la modalità di protezione)
unlink(name)	Cancella un file (assumendo che ci sia un solo link a esso)
open(name, mode)	Apre o crea un file e restituisce il suo descrittore
close(fd)	Chiude un file
read(fd, buffer, count)	Legge *count* byte in un *buffer*
write(fd, buffer, count)	Scrive *count* byte da un *buffer*
lseek(fd, offset, w)	Sposta il puntatore del file secondo *offset* e *w*
stat(name, buffer)	Restituisce le informazioni sul file
chmode(name, mode)	Cambia la modalità di protezione del file
fcntl(fd, cmd, ...)	Esegue varie operazioni di controllo come il lock di (parte di) un file

Figura 6.34 Chiamate principali per il file system di UNIX.

ne sono proprietari. Infine `fcntl` svolge svariate operazioni, tra cui l'aggiunta o la rimozione di un blocco che indica se il file è stato riservato o meno per l'accesso esclusivo da parte di un processo.

La Figura 6.35 illustra il funzionamento delle principali chiamate di I/O; il codice raffigurato è ridotto, in quanto non comprende il pur necessario controllo degli errori. Prima di entrare nel ciclo, il programma apre un file esistente, `data`, e ne crea uno nuovo, `newf`; le due chiamate prendono in ingresso come secondo parametro i bit di protezione che specificano la modalità dei due file, rispettivamente in lettura e in scrittura, e restituiscono a loro volta due descrittori di file, `infd` e `outfd`. Se una delle due chiamate non dovesse andare a buon fine, restituirebbe un descrittore di file negativo, a indicare l'insuccesso.

La procedura `read` ha tre parametri: un descrittore di file, un buffer e un numero di byte. La `read` prova a leggere dal file il numero di byte specificato e a porlo nel buffer, ma il numero di byte effettivamente letto è restituito in `count` e sarà minore di `bytes` se il file è troppo corto. La chiamata `write` ripone i byte appena letti nel file di output. Il ciclo prosegue finché non termina la lettura di tutto il file di input, al che il ciclo termina e i file vengono chiusi.

I descrittori di file di UNIX sono dei numeri interi (in genere minori di 20): i descrittori 0, 1 e 2 sono speciali e riservati rispettivamente a **standard input**, **standard output** e **standard error,** che puntano di norma alla tastiera, allo schermo e ancora allo schermo, ma che possono essere rediretti[1] su file specificati dall'utente. Molti programmi di UNIX prendono dati in ingresso dallo standard input e scrivono in uscita sullo standard output; spesso sono chiamati **filtri**.

Intimamente legato al file system è il directory system. Ogni utente può possedere molte directory, che a loro volta possono contenere file o altre sottodirectory. I sistemi UNIX sono configurati spesso come mostrato dalla Figura 6.36, con una directory principale, la **directory**

[1] Per redirezione di un puntatore (descrittore di file) si intende l'attribuzione a esso di una nuova locazione (direzione) a cui puntare (*N.d.T.*).


```
/* Apertura dei descrittori di file. */
infd = open("data", 0);
outfd = creat("newf", ProtectionBits);

/* Ciclo di copia. */
do {
      count = read(infd, buffer, bytes);
      if (count > 0) write(outfd, buffer, count);
} while (count > 0);

/* Chiusura dei file. */
close(infd);
close(outfd);
```

Figura 6.35 Frammento di programma per la copia di un file con chiamate di sistema UNIX. Usiamo il C perché mette bene in evidenza le chiamate di sistema di livello più basso (a differenza di Java, che le nasconde).

radice (*root directory*), che contiene le sottodirectory *bin* (per i programmi di esecuzione frequente), *dev* (per i file speciali di dispositivi di I/O), *lib* (per le librerie) e *usr* (per le directory degli utenti). Nell'esempio la directory *usr* contiene le sottodirectory *ast* e *jim*; la prima contiene a sua volta due file, *data* e *foo.c*, e una sottodirectory, *bin*, che contiene quattro giochi.

Un file è rintracciabile tramite il suo **percorso** (*path*) a partire dalla radice. Un percorso contiene la lista di tutte le directory attraversate dalla radice fino al file, separate da un carattere barra. Per esempio, il percorso assoluto del file *gioco2* è */usr/ast/bin/gioco2*. Si dice **percorso assoluto** un percorso di file che comincia dalla radice.

A ciascun programma in esecuzione è associata una **directory di lavoro** (*working directory*). I percorsi dei file possono essere specificati relativamente alla directory di lavoro, nel qual caso cominciano con una barra, perché siano distinguibili dai percorsi assoluti. Parliamo in tal caso di **percorsi relativi**: se la directory di lavoro è */usr/ast*, *gioco3*, il file è raggiungibile usando il percorso relativo *bin/gioco3*. Un utente può creare un **collegamento** (*link*) a un file di un altro utente grazie alla chiamata di sistema `link`. Nell'esempio precedente */usr/ast/bin/gioco3* e */usr/jim/jotto* puntano allo stesso file. Non sono permessi collegamenti a directory per evitare la formazione di cicli nel directory system. Le chiamate `open` e `creat` accettano come parametro sia percorsi relativi sia assoluti.

Le principali chiamate di sistema per la gestione delle directory UNIX sono elencate nella Figura 6.37. `mkdir` crea una nuova directory e `rmdir` rimuove una directory (vuota) già esistente. Le successive tre chiamate si usano per leggere gli elementi di una directory: la prima apre la directory, la seconda legge i suoi elementi, la terza chiude la directory. `chdir` cambia la directory di lavoro.

`link` crea in una directory un nuovo elemento che punta a un file esistente. Per esempio, l'elemento */usr/jim/jotto* potrebbe essere stato creato dalla chiamata

```
link("/usr/ast/bin/gioco3", "/usr/jim/jotto")
```

o da una chiamata equivalente con, in ingresso, percorsi relativi dipendenti dalla directory di lavoro del programma che effettua la chiamata. `unlink` rimuove un elemento dalla directory: se il file ha un solo collegamento viene cancellato, ma se ne ha di più viene mantenuto. Non ha alcuna importanza se il collegamento rimosso era l'originale o una copia.

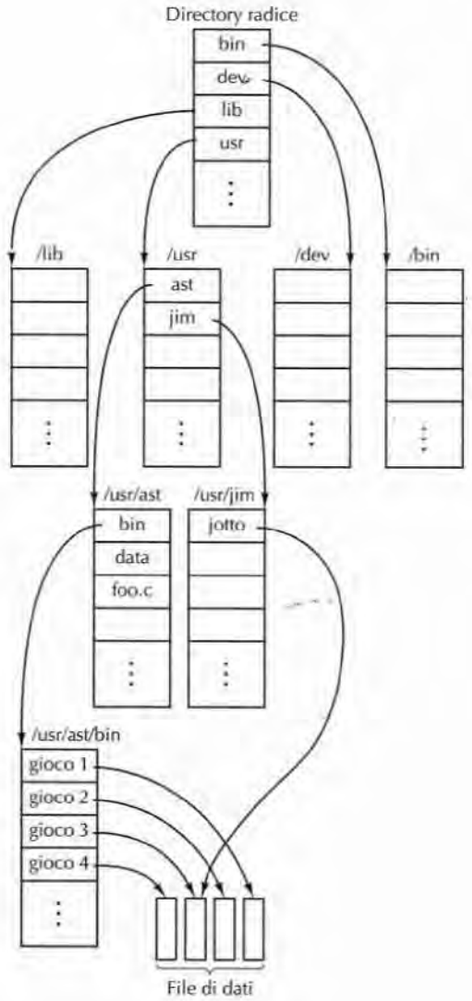


Figura 6.36 Parte del directory system di un tipico sistema UNIX.

Una volta creato, un collegamento ha tutti i diritti e le proprietà dell'originale, da cui è indistinguibile. La chiamata

```
unlink("/usr/ast/bin/gioco3")
```

rende *gioco3* accessibile d'ora in poi solo attraverso il percorso */usr/jim/jotto*. In questo modo è possibile usare `link` e `unlink` per "trasferire" file da una directory all'altra.

A ogni file (e alle directory, che sono esse stesse dei file) è associata una stringa di bit che specifica chi può accedervi, ed è costituita da tre campi RWX: il primo controlla i permessi si lettura (R), scrittura (W) ed esecuzione (X) del proprietario, il secondo controlla i permessi degli altri utenti dello stesso gruppo, il terzo riguarda tutti gli altri utenti. Così RWX R-X --X vuol dire che il proprietario può leggere, scrivere ed eseguire il file (se si tratta di un programma eseguibile, altrimenti la X sarebbe assente), mentre gli altri utenti del suo gruppo possono solo leggerlo o eseguirlo e gli estranei possono solo eseguirlo. Con questi permessi gli estranei possono usare il programma, ma non rubarlo (copiarlo), poiché non hanno il permesso in lettura. L'attribuzione di un gruppo agli utenti è fatta dall'amministratore di sistema, chiamato in genere **superuser**, che ha anche il potere di scavalcare il meccanismo di protezione e può leggere, scrivere o eseguire qualunque file.

Chiamata di sistema	Significato
mkdir(name, mode)	Crea una directory nuova
rmdir(name)	Elimina una directory vuota
opendir(name)	Apre una directory per la lettura
readdir(dirpointer)	Legge l'elemento successivo di una directory
closedir(dirpointer)	Chiude una directory
chdir(dirname)	Cambia la directory di lavoro in *dirname*
link(name1, name2)	Crea un elemento di directory *name2* che punta a *name1*
unlink(name)	Rimuove *name* dalla sua directory

Figura 6.37 Chiamate principali di UNIX per la gestione delle directory.

Veniamo ora brevemente all'implementazione dei file e delle directory UNIX (per una trattazione completa si veda Vahalia, 1996). A ogni file (e quindi a ogni directory) è associato un blocco d'informazioni di 64 byte detto **i-node**, che contiene le informazioni sul proprietario e sui permessi del file, sulla locazione dei dati e altre cose simili. Gli i-node dei file sono conservati in sequenza all'inizio del disco o all'inizio di un gruppo di cilindri, se il disco è così suddiviso. UNIX è in grado di calcolare l'indirizzo su disco di un i-node a partire dal suo numero nella sequenza.

Gli elementi delle directory consistono in due parti: un nome di file e un numero di i-node. Quando un programma esegue

```
open("foo.c", 0)
```

il sistema cerca il file di nome "foo.c" nella directory di lavoro al fine di localizzarne il numero di i-node. Una volta trovatone il numero può accedere all'i-node e a tutte le informazioni sul file.

Se viene specificato un percorso più lungo, i passi precedenti vengono ripetuti finché non sia stato attraversato l'intero percorso. Per esempio, per localizzare il numero di i-node di */usr/ast/data*, il sistema cerca prima l'elemento *usr* nella directory radice. Quando ha trovato l'i-node di *usr* può leggere il file corrispondente (ripetiamo: in UNIX le directory sono file) e cercare al suo interno l'elemento *ast*, arrivando a localizzare il numero di i-node del file */usr/ast*. La lettura di */usr/ast* consente al sistema di trovare l'elemento *data* e quindi il numero di i-node di */usr/ast/data*. Grazie a quel numero è possibile trovare tutto quanto riguarda il file suddetto.

Il formato, i contenuti e l'organizzazione degli i-node variano leggermente da sistema a sistema (specie se il sistema svolge attività di rete), ma almeno le voci seguenti si trovano in tutte le versioni di i-node:

1. il tipo del file, i 9 bit di protezione RWX e qualche altro bit
2. il numero di collegamenti al file (numero di elementi di directory che lo punta)
3. l'identità del proprietario
4. il gruppo del proprietario
5. la lunghezza del file in byte
6. tredici indirizzi su disco
7. la data dell'ultimo accesso in lettura
8. la data dell'ultimo accesso in scrittura
9. la data dell'ultima modifica all'i-node.

Riguardo il tipo, un file può essere un file ordinario, una directory o uno dei due tipi di file speciali che si usano per distinguere i dispositivi di I/O strutturati a blocchi da quelli non strutturati. Abbiamo già analizzato il numero di collegamenti e l'identificativo del proprietario. La lunghezza del file è data da un intero di 32 bit che specifica l'ultima locazione del file con un contenuto informativo: è del tutto lecito creare un file, fare una `lseek` alla posizione 1.000.000 e scrivere un byte, nel qual caso la lunghezza del file è di 1.000.001 byte. Va detto però che l'eventuale memorizzazione del file non richiederebbe alcuno spazio per i byte "mancanti".

I primi 10 indirizzi del disco puntano a blocchi di dati. Se la dimensione di un blocco è 1024 byte, in questo modo è possibile gestire file fino alla dimensione di 10.240 byte. L'indirizzo 11 punta a un blocco del disco che si chiama **blocco indiretto**, contenente 256 indirizzi del disco. In questo modo è possibile gestire file fino alla dimensione di 10.240 + 256 × 1024 = 272.384 byte. Nel caso dei file ancora più grandi si usa l'indirizzo 12 che punta a un blocco contenente 256 blocchi indiretti, riuscendo a gestire così file di 272.384 + 256 × 256 × 1024 = 67.381.248 byte. Se anche questo schema con **blocco indiretto doppio** non è sufficiente per contenere il file, si usa l'indirizzo 13 per puntare a un **blocco indiretto triplo** contenente 256 indirizzi di blocchi indiretti doppi. Usando congiuntamente gli indirizzi dei blocchi diretti, indiretti, indiretti doppi e tripli si possono indirizzare 16.843.018 blocchi, raggiungendo il limite teorico di dimensione per i file di 17.247.250.432 byte. In realtà, essendo i puntatori di file di 32 bit, il limite pratico è di 4.294.967.295 byte. I blocchi liberi del disco sono mantenuti in una lista concatenata da cui vengono rimossi uno alla volta quando necessari. Di conseguenza, i blocchi di ciascun file si trovano sparpagliati alla rinfusa per tutto il disco.

Al fine di rendere le operazioni di I/O più efficienti, all'apertura di un file il suo i-node viene copiato (per ragioni di comodità) in una tabella in memoria principale e lì mantenuto finché il file non viene chiuso. Inoltre viene tenuta in memoria anche una raccolta dei blocchi referenziati di recente; poiché i file vengono letti in modo sequenziale nella stragrande maggioranza dei casi, succede spesso che un riferimento alla locazione di un

file si trovi all'interno dello stesso blocco del riferimento precedente. Motivato da questa considerazione, il sistema cerca di leggere anche il blocco *successivo* a quello correntemente in lettura prima che venga effettivamente referenziato, in modo da accelerare l'esecuzione. Tutti questi meccanismi di ottimizzazione sono nascosti all'utente: quando un utente effettua una `read`, il suo programma resta sospeso finché non siano disponibili nel buffer i dati da lui richiesti.

Alla luce di queste nozioni, possiamo ora capire come funziona l'I/O su file. La `open` induce il sistema a cercare le directory nel percorso specificato: se la ricerca ha successo, l'i-node viene caricato nella tabella interna. Le chiamate `read` e `write` richiedono al sistema il calcolo del numero di blocco a partire dalla posizione corrente nel file. Gli indirizzi su disco dei primi 10 blocchi si trovano sempre in memoria centrale (nell'i-node); i blocchi di numero maggiore richiedono la lettura preventiva di uno o più blocchi indiretti. `lseek` si limita a spostare il puntatore alla posizione corrente senza effettuare alcun I/O.

È ora semplice comprendere anche `link` e `unlink`: `link` cerca all'interno del primo argomento il numero di i-node, quindi crea un elemento di directory nel secondo argomento, assegnandogli lo stesso numero di i-node. Infine incrementa di un'unità il numero di collegamenti nell'i-node. `unlink` rimuove un elemento da una directory e decrementa il numero di collegamenti nell'i-node. Se il numero arriva a zero, il file viene cancellato e i suoi blocchi resi disponibili tramite la loro inserzione nella lista dei blocchi liberi.

I/O virtuale di Windows XP

XP supporta molti file system, tra cui i più importanti sono **NTFS** (*NT File System*) e **FAT** (*File Allocation Table*, "tabella di allocazione dei file"). NTFS è stato sviluppato specificamente per XP; FAT è il vecchio file system di MS-DOS, usato anche da Windows 95/98 (anche se con l'introduzione dei nomi di file lunghi). Poiché FAT è ormai obsoleto ci limitiamo allo studio del primo.

I nomi di file in NTFS possono essere lunghi un massimo di 255 caratteri e utilizzano Unicode per consentire agli utenti di quei paesi la cui lingua non deriva dal latino (per esempio i giapponesi, gli indiani o gli israeliani), di attribuire ai file nomi nella loro lingua madre. Per la precisione, XP usa Unicode estesamente al proprio interno; fin da Windows 2000 il sistema viene distribuito in un unico codice binario e le differenze linguistiche, che si manifestano nei menu, nei messaggi di errore, e così via, sono gestite tramite file di configurazione differenziati per nazione. NTFS è un sistema completamente *case sensitive* (cioè, per esempio, *pippo* è diverso da *PIPPO*) ma sfortunatamente le API di Win32 lo sono solo parzialmente per i nomi di file e non lo sono affatto per i nomi di directory, così questa proprietà viene a mancare ai programmi che usano Win32.

Come per UNIX, un file XP è semplicemente una sequenza lineare di byte, che però può raggiungere i $2^{64} - 1$ byte. Pertanto i puntatori di file, che esistono come in UNIX, sono di 64 bit invece che di 32 per gestire la massima lunghezza possibile. Le chiamate di funzioni API di Win32 per la manipolazione dei file e delle directory sono abbastanza simili alla loro controparte UNIX, a parte il fatto che hanno spesso più parametri e che si avvalgono di un modello di sicurezza diverso. L'apertura di un file restituisce un handle, poi usato per la lettura o la scrittura del file. A differenza di UNIX, però, gli handle non sono piccoli interi, e lo standard input, lo standard output e lo standard error non sono predefiniti come 0, 1 e 2, bensì devono essere acquisiti esplicitamente (tranne che in modalità con-

sole, in cui sono già aperti). Le funzioni principali delle API di Win32 per la gestione dei file sono elencate nella Figura 6.38.

Veniamo ora alle chiamate di sistema. `CreateFile` genera un file nuovo e restituisce il suo handle; si usa anche per aprire i file già esistenti, dato che non esiste una funzione *open* specifica. Nella figura non abbiamo incluso i parametri delle funzioni API di XP perché sono molto voluminosi. Solo per fare un esempio, i parametri della `CreateFile` sono sette:

1. un puntatore al nome del file da creare o da aprire
2. dei flag per indicare se il file può essere letto, scritto, o entrambe le cose
3. dei flag che specificano se il file può essere aperto contemporaneamente da più processi
4. un puntatore al descrittore di sicurezza che stabilisce chi può accedere al file
5. dei flag che indicano il comportamento da assumere se il file esiste/non esiste
6. dei flag che riguardano gli attributi del file, come il suo essere un file di archivio, compresso, e così via
7. l'handle di un file da cui clonare gli attributi per l'attribuzione al nuovo file.

Le sei funzioni API successive nella Figura 6.38 sono abbastanza simili alle corrispondenti chiamate di UNIX. Le ultime due consentono di porre o di togliere il lock su una regione di un file per garantirne l'accesso esclusivo da parte di un processo (ovvero la mutua esclusione tra processi).

A partire da queste funzioni API è possibile scrivere una procedura per la copia di un file analoga a quella della Figura 6.35. La Figura 6.39 mostra una siffatta procedura (senza controllo degli errori), concepita sul modello della versione UNIX. Nella pratica non c'è alcun bisogno di usare il codice della figura, perché esiste la funzione API `CopyFile` che chiama una procedura di libreria il cui codice è simile a quello indicato nell'esempio.

XP supporta un file system gerarchico simile a quello di UNIX, anche se il carattere separatore dei componenti di un nome è \ invece di /, un retaggio di MS-DOS. Esiste il

Funzione API	UNIX	Significato
CreateFile	open	Crea un file o apre un file esistente; restituisce l'handle
DeleteFile	unlink	Cancella un file esistente
CloseHandle	close	Chiude un file
ReadFile	read	Legge dati da un file
WriteFile	write	Scrive dati in un file
SetFilePointer	lseek	Assegna al puntatore del file una determinata posizione
GetFileAttributes	stat	Restituisce le informazioni sul file
LockFile	fcntl	Mette in lock una regione del file per garantire la mutua esclusione
UnlockFile	fcntl	Toglie il lock da una regione di un file precedentemente riservata

Figura 6.38 Funzioni principali delle API di Win32 per l'I/O di file. La seconda colonna riporta la funzione di UNIX più simile.


```
/* Apertura dei file per input e output. */
inhandle = CreateFile("data", GENERIC_READ, 0, NULL, OPEN_EXISTING, 0, NULL);
outhandle = CreateFile("newf", GENERIC_WRITE, 0, NULL, CREATE_ALWAYS,
        FILE_ATTRIBUTE_NORMAL, NULL);

/* Copia del file. */
do {
        s = ReadFile(inhandle, buffer, BUF_SIZE, &count, NULL);
        if (s > 0 && count > 0) WriteFile(outhandle, buffer, count, &ocnt, NULL);
} while (s > 0 && count > 0);

/* Chiusura dei file. */
CloseHandle(inhandle);
CloseHandle(outhandle);
```

Figura 6.39 Frammento di programma per la copia di un file con le funzioni API di Windows XP. Usiamo il C perché mette bene in evidenza le chiamate di sistema di livello più basso (a differenza di Java, che le nasconde).

concetto di directory di lavoro e i nomi di percorso possono essere assoluti o relativi. C'è comunque una differenza sostanziale rispetto a UNIX: UNIX permette di "montare" (*mount*) insieme i file system di dischi o di macchine diversi in un unico albero dei nomi, nascondendo così la struttura dei dischi al software. XP 4.0 non presenta questa proprietà, perciò i nomi di file assoluti devono cominciare con la lettera del drive (l'unità) che indica il disco logico in cui si trova il file, per esempio in *C:\windows\system\foo.dll*. A partire da Windows 2000 è stata introdotta la possibilità di montare un file system nello stile di UNIX.

La Figura 6.40 elenca le funzioni API più importanti per la gestione delle directory insieme alle corrispettive funzioni UNIX. I loro nomi sono sufficientemente autoesplicativi.

XP è dotato di un meccanismo di sicurezza molto più elaborato di quello di molti sistemi UNIX. Benché esistano centinaia di funzioni API relative alla sicurezza, cerchiamo di fornire una breve descrizione dell'idea generale. Quando un utente effettua il login, il sistema operativo affida al suo processo iniziale un **gettone di accesso** (*access token*) che contiene il **SID** (*Security ID*) dell'utente, una lista dei gruppi di sicurezza cui egli appartiene, i privilegi speciali di cui dispone e poche altre informazioni. L'idea del gettone di accesso è di concentrare tutte le informazioni di sicurezza in un solo posto facile da reperire. Tutti i processi creati a partire da quello iniziale ereditano lo stesso gettone di accesso.

Uno dei parametri d'ingresso che può essere passato all'atto della creazione di un oggetto è il suo **descrittore di sicurezza**, contenente una lista di controllo degli accessi, **ACL** (*Access Control List*). Ogni elemento della lista consente oppure vieta un certo insieme di operazioni sull'oggetto da parte di un certo SID o di un determinato gruppo. Per esempio un file potrebbe avere un descrittore di sicurezza secondo cui Marta non ha accesso al file, Marco può leggerlo soltanto, Linda può leggerlo o scrivervi e tutti i membri del gruppo XYZ possono soltanto leggere la lunghezza del file, nient'altro.

Quando un processo cerca di eseguire un'operazione su un oggetto tramite l'handle restituitogli dall'apertura dell'oggetto stesso, il gestore della sicurezza recupera il gettone di accesso del processo e scorre gli elementi della ACL uno per volta. Non appena incontra un elemento corrispondente al SID o a gruppo del chiamante, il tipo di accesso ivi consentito è considerato come definitivo. Per questo motivo si usa elencare nella ACL prima gli ele-

Funzione API	UNIX	Significato
CreateDirectory	mkdir	Crea una directory nuova
RemoveDirectory	rmdir	Elimina una directory vuota
FindFirstFile	opendir	Prepara alla lettura degli elementi di una directory
FindNextFile	readdir	Legge l'elemento di directory successivo
MoveFile		Sposta un file da una directory in un'altra
SetCurrentDirectory	chdir	Cambia la directory di lavoro corrente

Figura 6.40 Principali funzioni API di Win32 per la gestione di directory. La seconda colonna indica la funzione di UNIX più simile, se ne esiste una.

menti che negano l'accesso, poi quelli che lo consentono, così che un utente cui è stato negato espressamente l'accesso non riesca a rientrare dalla porta di servizio perché appartenente a un gruppo autorizzato. Il descrittore di sicurezza contiene anche informazioni per la certificazione degli accessi all'oggetto.

Diamo ora uno sguardo all'implementazione di file e directory XP. Ogni disco è suddiviso in volumi statici indipendenti che hanno la stessa funzione delle partizioni UNIX. Ogni volume contiene una bit map, file e directory, più altre strutture dati per la gestione delle informazioni. Ciascun volume è organizzato come una sequenza lineare di **cluster** ("raggruppamenti") di dimensione prefissata (compresa tra i 512 byte e i 64 KB) che dipende dalla dimensione del volume stesso. I riferimenti a un cluster avvengono tramite il suo offset dall'inizio del volume, per il quale si usa un numero di 64 bit.

La struttura dati principale di un volume è la **MFT** (*Master File Table*, "tabella principale di allocazione dei file") che contiene un elemento per ogni file o directory del volume, analogo all'i-node di UNIX. La stessa MFT è un file e perciò può essere posizionata ovunque nel volume, risolvendo così il problema di UNIX relativo alla presenza di blocchi del disco corrotti (rovinati) all'interno degli i-node.

La MFT (illustrata nella Figura 6.41) comincia con un'intestazione contenente le informazioni relative al volume, come il puntatore alla directory di radice, il file di boot, il file dei blocchi corrotti, l'amministrazione della lista delle locazioni libere, e così via. Di seguito si trovano gli elementi relativi ai file o alle directory che occupano 1 KB ciascuno, tranne quando la dimensione del cluster è di 2 KB o più. Ogni elemento contiene i metadati (informazioni amministrative) che riguardano il file o la directory. Sono consentiti diversi formati, uno dei quali è presentato nella Figura 6.41.

Il campo delle informazioni standard contiene alcuni dati come le etichette temporali (*time stamp*) necessarie al computo del numero di collegamenti, i bit di sola lettura o di archivio, e così via. Questo campo ha lunghezza fissa ed è sempre presente tra i metadati. Il campo del nome del file contiene un massimo di 255 caratteri Unicode. Affinché i file siano ancora accessibili dai programmi a 16 bit, viene fornito anche un campo per il nome MS-DOS costituito da otto caratteri alfanumerici eventualmente seguiti da un punto e da non più di tre caratteri alfanumerici per l'estensione. Se il vero nome del file rispetta già lo standard MS-DOS 8+3, non viene fornito un secondo nome MS-DOS.

Il campo successivo contiene le informazioni di sicurezza. Fino alla versione di XP 4.0 questo campo conteneva il descrittore di file vero e proprio. A partire da Windows 2000 si

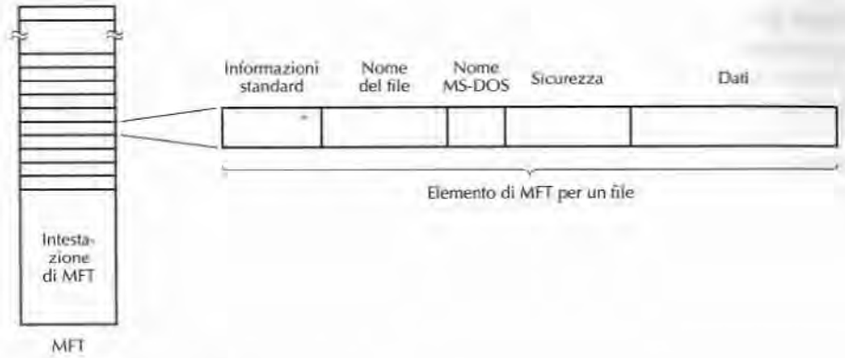


Figura 6.41 Tabella MFT di Windows XP.

è deciso di accentrare tutte le informazioni di sicurezza in un file unico e lasciare nel campo sicurezza un puntatore a una locazione del file.

Il campo dei dati nell'elemento di MFT contiene il file vero e proprio se questo è abbastanza piccolo, risparmiando così un accesso a disco. Questo stratagemma si chiama **file immediato** (Mullender e Tanenbaum, 1984). Per quanto riguarda i file più grandi, il campo dati contiene i puntatori ai cluster contenenti i dati oppure, più comunemente, a intere sequenze di cluster consecutivi; così con un solo identificativo di cluster e un parametro di lunghezza è possibile specificare una quantità di dati arbitraria. Se un elemento di MFT dovesse rivelarsi insufficiente a contenere tutte le informazioni richieste da un file è sempre possibile accorparlo a uno o più elementi successivi della tabella.

La dimensione massima dei file è di 2^{64} byte. Per capire quanto grande sia 2^{64} byte, si immagini una sequenza di zeri e di uno che occupino ciascuno un millimetro: una sequenza di 2^{67} bit siffatti (2^{64} byte) sarebbe lunga 15 anni luce, abbastanza per oltrepassare i confini del sistema solare, raggiungere Alpha Centauri e tornare indietro.

Il file system NTFS presenta molte altre proprietà interessanti, tra cui la compressione dei dati e la tolleranza agli errori ottenuta per mezzo delle transazioni atomiche. Per informazioni ulteriori riguardo NTFS si consulti (Russinovich e Solomon 2005).

6.4.4 Esempi di gestione dei processi

UNIX e XP prevedono entrambi la possibilità di suddividere un compito (*job*) tra più processi che possono girare in modo (pseudo)parallelo e comunicare con lo schema produttore-consumatore già illustrato. In questo paragrafo analizziamo la gestione dei processi dei due sistemi. Vedremo anche che entrambi supportano il parallelismo all'interno di un singolo processo mediante i thread.

Gestione dei processi in UNIX

Un processo UNIX può in ogni istante usare la chiamata di sistema `fork` per creare un sottoprocesso che è una sua esatta replica. Il processo originale è detto **genitore** e quello nuovo si chiama processo **figlio**. Appena dopo la `fork`, i due processi sono identici e condividono

addirittura gli stessi descrittori di file. Di lì in poi ciascuno prosegue per la propria strada e svolge il proprio compito indipendentemente dall'altro.

Spesse volte il processo figlio manipola i descrittori di file ed esegue una chiamata di sistema `exec` che rimpiazza il suo programma e i suoi dati con il programma e i dati contenuti in un certo file eseguibile specificato come parametro della chiamata `exec`. Per esempio, quando un utente immette il comando *xyz* in un terminale, l'interprete a linea di comando (la shell) esegue una `fork` per creare un processo figlio che svolga il programma *xyz* a seguito di una `exec`.

I due processi girano in parallelo (con o senza `exec`) a meno che il genitore voglia attendere la terminazione del figlio prima di continuare. In questa evenienza, il genitore esegue una chiamata di sistema `wait` o `waitpid` che causa la sua sospensione fino al termine dell'esecuzione del figlio. Solo allora il genitore riprende l'esecuzione.

I processi possono eseguire quante `fork` vogliono, dando vita così a un albero di processi. La Figura 6.42 mostra il processo *A* che ha eseguito due `fork` e ha così generato i due figli *B* e *C*. Anche *B* ha eseguito due volte una `fork`, mentre *C* l'ha eseguita una volta sola, per un totale di sei processi.

In UNIX i processi possono comunicare tra loro attraverso una struttura chiamata **pipe** ("conduttura"). Una pipe è una specie di buffer in cui un processo può scrivere un flusso di dati disponibili al prelievo da parte di un altro processo. I byte sono prelevati da una pipe sempre nell'ordine in cui sono stati inseriti, non è mai consentito l'accesso diretto a una certa posizione. Le pipe non preservano le estremità dei messaggi, perciò se un processo effettua quattro scritture da 128 byte e un altro processo effettua una lettura da 512 byte, il secondo riceverà tutti i dati in una volta sola, senza alcuna traccia del fatto che sono stati scritti in più operazioni.

System V e Solaris mettono a disposizione un'altra modalità di comunicazione tra processi che usa le **code di messaggi** (*message queue*). Un processo può usare la chiamata `msgget` per creare una nuova coda di messaggi o aprirne una esistente, può invocare `msgsnd` o `msgrecv` per spedire o ricevere un messaggio su una coda. I messaggi che passano in una coda differiscono per molti versi da quelli che attraversano le pipe. Innanzitutto vengono preservati gli estremi dei messaggi, mentre una pipe è solo un flusso di dati. Poi i messaggi possono avere priorità, così quelli urgenti possono sopravanzare quelli meno importanti. Infine, i messaggi hanno un tipo, che può essere specificato tramite la `msgrecv`.

Un altro meccanismo per la comunicazione tra processi è la condivisione di una o più regioni dei loro spazi degli indirizzi. UNIX gestisce la condivisione di memoria attraverso la

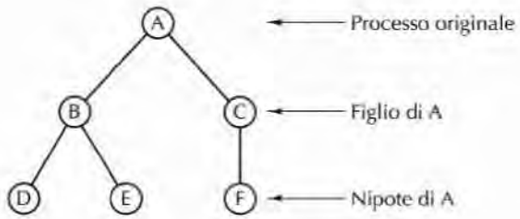


Figura 6.42 Albero di processi in UNIX.

corrispondenza simultanea tra alcune pagine e gli spazi degli indirizzi di tutti i processi. L'effetto di questa soluzione è che una scrittura operata in una regione condivisa è immediatamente visibile agli altri processi. Questo meccanismo garantisce una comunicazione con larghezza di banda molto elevata. Le chiamate di sistema coinvolte nella condivisione di memoria si chiamano `shmat` e `shmop`.

Un'altra caratteristica di System V e di Solaris è il supporto dei semafori, il cui funzionamento ricalca in buona sostanza quanto visto nell'esempio del produttore-consumatore.

Tutti i sistemi UNIX forniscono la gestione di thread multipli all'interno di un singolo processo. I **thread** sono processi leggeri che condividono lo stesso spazio degli indirizzi e tutto ciò che gli è associato: i descrittori di file, le variabili d'ambiente e i timer esterni. Tuttavia, ogni thread ha il proprio program counter, i propri registri e il proprio stack. Quando un thread si blocca (per esempio perché si trova in attesa del completamento di un'operazione di I/O o di un altro evento) l'esecuzione può passare agli altri thread dello stesso processo. Due thread produttore-consumatore che operano all'interno dello stesso processo sono simili, ma non identici, a due processi costituiti ciascuno da un solo thread e che condividono un segmento di memoria come buffer. Le differenze stanno nel fatto che in quest'ultimo caso ogni processo ha i propri descrittori di file, variabili d'ambiente, e così via, mentre nel primo caso questi enti sono condivisi. Abbiamo incontrato i thread Java nell'esempio del produtttore-consumatore. Spesso il sistema esecutivo di Java si avvale dei thread del sistema operativo per i propri thread, ma può anche scegliere di non farlo.

I server web sono un buon esempio per illustrare l'utilità dei thread. Questi server possono contenere una cache della pagine web usate più comunemente, così se arriva la richiesta di una pagina presente nella cache, questa può essere inviata immediatamente. In caso contrario, la pagina deve essere caricata dal disco e ciò comporta tempi maggiori (20 ms circa), durante i quali il processo resta bloccato e non può servire nuove richieste in ingresso, anche se si trattasse di richieste di pagine nella cache.

La soluzione consiste nel generare più thread all'interno del processo server, dove tutti condividono la stessa cache delle pagine web. Quando si blocca un thread, gli altri possono continuare a gestire le richieste in ingresso. In alternativa, sarebbe possibile moltiplicare il numero dei processi invece di quello dei thread, ma ciò richiederebbe probabilmente una replica della cache per ogni processo, con conseguente spreco di memoria.

Lo standard di UNIX per i thread si chiama **pthread** ed è definito dal documento P1003.1C. Sono previste chiamate per la gestione e per la sincronizzazione di thread, ma non è specificato se questi devono risiedere nello spazio del kernel o in quello dell'utente. Le chiamate più comuni per la gestione dei thread sono elencate nella Figura 6.43.

La prima chiamata, `pthread_create`, crea un nuovo thread, e dopo il suo completamento lo spazio degli indirizzi del chiamante contiene un thread in più in esecuzione. Quando un thread ha completato il proprio compito chiama `pthread_exit`, che ne provoca la terminazione. Un thread può attendere la terminazione di un altro thread tramite la chiamata `pthread_join`: se il thread che si vuole attendere è già terminato la chiamata si interrompe immediatamente, altrimenti il chiamante resta bloccato in attesa.

I thread si sincronizzano grazie a particolari lock chiamati **mutex** e usati generalmente per sorvegliare l'uso di buffer condivisi da due thread. Per avere la certezza di essere l'unico ad accedere a una risorsa condivisa, un thread deve assicurarsi il mutex prima di modificare

Chiamata di thread	Significato
pthread_create	Crea un nuovo thread nello spazio degli indirizzi del chiamante
pthread_exit	Provoca la terminazione del thread chiamante
pthread_join	Aspetta la terminazione di un thread
pthread_mutex_init	Crea un nuovo mutex
pthread_mutex_destroy	Distrugge un mutex
pthread_mutex_lock	Riserva un mutex
pthread_mutex_unlock	Rilascia un mutex
pthread_cond_init	Crea una variabile di condizione
pthread_cond_destroy	Distrugge una variabile di condizione
pthread_cond_wait	Attende su una variabile di condizione
pthread_cond_signal	Libera un thread in attesa su di una variabile di condizione

Figura 6.43 Principali chiamate POSIX per i thread.

la risorsa e deve rimuoverlo non appena ha concluso il proprio accesso. Se tutti i thread seguissero questo protocollo, evitare le corse critiche sarebbe possibile. I mutex somigliano a semafori binari, cioè a semafori che possono assumere solo i valori 0 e 1. Il nome "mutex" deriva dal fatto che sono usati per assicurare la mutua esclusione (*mutual exclusion*) da certe risorse.

Le chiamate `pthread_mutex_init` e `pthread_mutex_destroy` servono rispettivamente a creare e a distruggere un mutex. Un mutex può trovarsi in uno di due stati: riservato o non riservato. Per riservare un mutex non riservato, un thread usa `pthread_mutex_lock` che ha effetto immediato; viceversa, se il mutex è già riservato la chiamata diventa bloccante. Spetta al thread che aveva riservato il mutex precedentemente chiamare `pthread_mutex_unlock` per rilasciarlo non appena abbia completato le proprie operazioni sulla risorsa condivisa.

I mutex servono a salvaguardare gli accessi a breve termine alle variabili condivise, mentre per la sincronizzazione a lungo termine (come nell'attesa per un'unità a nastro) si usano le **variabili di condizione**. La loro creazione e distruzione avviene tramite le chiamate `pthread_cond_init` e `pthread_cond_destroy`.

L'uso di una variabile di condizione coinvolge due thread, uno in attesa sulla variabile e un altro che effettua la segnalazione per sopraggiunta disponibilità. Per esempio, se un thread realizza che l'unità a nastro che gli serve è occupata, chiama `pthread_cond_wait` su una variabile di condizione associata all'unità e precedentemente concordata tra tutti i thread. Quando il thread che ha in uso l'unità completa la propria attività su di essa (non importa quanto tempo dopo), chiama `pthread_cond_signal` per liberare esattamente un thread in attesa su quella variabile di condizione (se ce n'è almeno uno). Se non c'è alcun thread in attesa il segnale va perso; le variabili di condizione non sono contatori come i semafori. Esistono poche altre operazioni definite sui thread, sui mutex e sulle variabili di condizione che però non tratteremo.

Gestione dei processi in Windows XP

XP supporta esistenza, comunicazione e sincronizzazione di più processi. Ogni processo contiene almeno un thread, che a sua volta contiene almeno una *fibra* (thread leggero). Processi, thread e fibre costituiscono un insieme di strumenti molto generale per la gestione del parallelismo sia su sistemi uniprocessore (macchine con CPU singola) sia multiprocessore (macchine con più CPU).

I processi sono creati mediante la funzione API `CreateProcess` che accetta 10 parametri, ciascuno dotato di molte opzioni. Si tratta evidentemente di un progetto molto più complesso dello schema di UNIX, la cui `fork` non ha parametri, mentre `exec` ne ha solo tre: il puntatore al nome del file da eseguire, l'array dei parametri individuati nella linea di comando e le stringhe d'ambiente. Con buona approssimazione, i 10 parametri di `CreateProcess` servono a contenere:

1. un puntatore al nome del file eseguibile
2. la stessa linea di comando (prima dell'analisi di parsing)
3. un puntatore al descrittore di sicurezza del processo
4. un puntatore al descrittore di sicurezza del thread iniziale
5. un bit che determina se il nuovo processo eredita gli handle del processo creatore
6. vari flag (per esempio la modalità d'errore, la priorità, il debugging, le console)
7. un puntatore alle stringhe d'ambiente
8. un puntatore al nome della directory di lavoro del nuovo processo
9. un puntatore alla struttura che descrive la finestra iniziale a schermo
10. un puntatore alla struttura che serve per restituire 18 valori al chiamante.

XP non impone alcun rapporto di parentela o gerarchia tra processi: tutti i processi creati sono uguali. D'altra parte, poiché uno dei 18 parametri restituiti al chiamante è l'handle del nuovo processo (che gli consente un certo controllo sul processo neonato) risulta definita implicitamente una gerarchia in base alla relazione "il processo *x* possiede l'handle del processo *y*". Nonostante sia vietato il passaggio diretto di handle tra processi, c'è comunque un modo con cui un processo può adattare un handle per passarlo a un altro e così la gerarchia definita implicitamente può avere vita breve.

Ogni processo di XP all'atto della creazione contiene un solo thread, ma è possibile aggiungervene successivamente di nuovi. La creazione dei thread è più semplice di quella dei processi; `CreateThread` ha soli 6 parametri invece di 10: descrittore di sicurezza, dimensione dello stack, indirizzo iniziale, un parametro definito dall'utente, stato iniziale (pronto o bloccato) e ID. La creazione dei thread è svolta dal kernel che è dunque al corrente della loro esistenza (i thread non sono implementati interamente nello spazio dell'utente come avviene in altri sistemi).

Durante la propria attività di scheduling, il kernel non solo sceglie quale processo far eseguire, ma anche quale dei suoi thread. Ciò significa che il kernel è sempre infor-

mato sullo stato (pronto o bloccato) di ogni thread. I thread sono oggetti del kernel e in quanto tali hanno un handle e un descrittore di sicurezza. Poiché è possibile il passaggio di handle tra processi, è altresì possibile per un processo controllare i thread di un altro processo. È questa una caratteristica utile, per esempio, alla scrittura di debugger.

La gestione dell'alternanza tra i thread di XP è abbastanza costosa, perché ogni passaggio dall'esecuzione di un thread a un altro comporta l'ingresso e l'uscita dal kernel. XP mette a disposizione le **fibre** per fornire uno pseudoparallelismo leggero; sono simili ai thread, ma il loro scheduling viene effettuato nello spazio dell'utente a opera del programma che le ha create (o dal suo sistema runtime). Ogni thread può contenere più fibre, allo stesso modo di come un processo può contenere più thread, con la differenza però che quando una fibra si blocca si aggiunge da sola alla coda delle fibre bloccate e sceglie un'altra fibra dello stesso thread a cui passare l'esecuzione. Il kernel non si accorge di questa transizione perché resta in esecuzione, anche se l'esecuzione è passata da una fibra all'altra. Il kernel si cura solo dei processi e dei thread, non delle fibre. Le fibre si rivelano utili, per esempio, quando c'è bisogno di portare su XP un programma che gestisce da solo i propri thread.

I processi possono comunicare in un numero svariato di modi; attraverso pipe, pipe con nome, *mailslot*, socket, chiamate di procedura a distanza e file condivisi. Alla creazione di una pipe si può scegliere tra due modalità di funzionamento: a byte e a messaggi. Le pipe a byte funzionano come quelle di UNIX. Le pipe a messaggi sono abbastanza simili, ma preservano le estremità dei messaggi: così quattro scritture di 128 byte ciascuna verranno ricevute come quattro messaggi di 128 byte e non come un solo messaggio di 512 byte, come accadrebbe con una pipe a byte. Esistono anche le pipe con nome, che mettono a disposizione entrambe le modalità. A differenza delle pipe ordinarie, le pipe con nome possono essere usate anche via rete.

Una caratteristica di XP che non è presente in UNIX è data dalle **mailslot**, per certi versi simili alle pipe, ma non del tutto; a differenza delle pipe sono infatti monodirezionali. Inoltre possono essere usate via rete, ma non garantiscono la ricezione dell'informazione. Infine consentono al processo mittente di inviare il messaggio in modo broadcast a molti destinatari.

Le socket sono come le pipe, se non che di norma collegano processi che risiedono su macchine diverse, ma possono essere usate anche per la comunicazione tra processi della stessa macchina. In genere non assicurano vantaggi particolari rispetto alla connessione di due macchine tramite una pipe, con o senza nome.

Le chiamate di procedura a distanza (*remote procedure call*) consentono al processo *A* di richiedere al processo *B* di chiamare per proprio conto una terza procedura nello spazio degli indirizzi di *B*, e di farsi restituire il risultato. Esistono diverse limitazioni sui parametri; per esempio non ha alcun senso passare un puntatore a un altro processo.

Infine i processi possono condividere la memoria attraverso la sovrapposizione di uno stesso file. Tutte le scritture effettuate da un processo compaiono anche nello spazio degli indirizzi dell'altro processo. Con questo meccanismo l'implementazione del buffer condiviso nell'esempio del produttore-consumatore è molto semplice.

Così come XP fornisce molti meccanismi di comunicazione tra processi, mette anche a disposizione un gran numero di tecniche di sincronizzazione, tra cui i semafori, i mutex, le sezioni critiche e gli eventi. Sono tutti meccanismi che operano a livello dei thread, non dei processi, perciò se un thread si blocca su un semaforo ciò non ha alcun effetto sugli (eventuali) altri thread dello stesso processo, che possono così continuare la propria esecuzione.

La funzione API `CreateSemaphore` crea un nuovo semaforo, con la possibilità di assegnargli un certo valore e un valore massimo. I semafori sono oggetti del kernel e hanno quindi un handle e un descrittore di sicurezza. È possibile duplicare l'handle di un semaforo attraverso la chiamata `DuplicateHandle` e passarlo a un altro processo affinché diversi processi si sincronizzino sullo stesso semaforo. Sono presenti le chiamate per le operazioni di `up` e `down`, anche se hanno nomi un po' peculiari: `ReleaseSemaphore` e `WaitForSingleObject` rispettivamente. Inoltre è possibile passare a `WaitForSingleObject` un timeout dopo il quale il thread chiamante viene liberato in ogni caso, anche se il semaforo resta a 0 (e sebbene l'uso del timer reintroduca le corse critiche).

Anche i mutex sono oggetti del kernel usati per la sincronizzazione, ma sono più semplici dei semafori perché non sono associati a contatori. Sono fondamentalmente dei lock cui corrispondono le funzioni API `WaitForSingleObject` (per riservarli) e `ReleaseMutex` (per rilasciarli). Al pari degli handle dei semafori, anche gli handle dei mutex possono essere duplicati e passati tra processi di modo che processi diversi accedano agli stessi mutex.

Il terzo meccanismo di sincronizzazione si basa sulle **sezioni critiche**, simili ai mutex Diversamente da questi però, le sezioni critiche non sono oggetti del kernel, ma sono locali allo spazio degli indirizzi del thread che le ha generate. Di conseguenza non hanno handle né descrittori di sicurezza e non possono essere passate da un processo all'altro. Vengono riservate e rilasciate per mezzo delle chiamate `EnterCriticalSection` e `LeaveCriticalSection`. Sono funzioni molto più veloci delle analoghe definite sui mutex, perché vengono svolte interamente nello spazio dell'utente.

L'ultima forma di sincronizzazione usa ancora oggetti del kernel, che prendono il nome di **eventi**. Un thread può attendere che si verifichi un evento richiamando `WaitForSingleObject`, può liberare un thread in attesa su un evento per mezzo di `SetEvent`, o addirittura invocare `PulseEvent` per risvegliare tutti i thread in attesa su quell'evento. Anche gli eventi presentano molte varianti e consentono di scegliere tra diverse opzioni.

È possibile attribuire un nome agli eventi, ai mutex e ai semafori e memorizzarli nel file system, proprio come avviene per le pipe con nome. Due o più processi possono poi sincronizzarsi tramite la semplice apertura dello stesso evento, mutex o semaforo, senza il bisogno che un processo crei l'oggetto e ne duplichi l'handle per passarlo a tutti gli altri.

6.5 Riepilogo

Si può guardare al sistema operativo come al garante di alcune funzionalità architetturali non presenti nel livello ISA. Prime fra tutte ci sono la memoria virtuale, le istruzioni virtuali di I/O e il supporto del calcolo parallelo.

La memoria virtuale è una funzionalità architetturale che ha lo scopo di fornire ai programmi spazi d'indirizzi più grandi della memoria fisica del sistema, insieme a un meccanismo flessibile e corretto per la protezione e la condivisione della memoria. Può essere implementata con la paginazione pura, con la segmentazione pura o con una combinazione delle due. Nel primo caso lo spazio degli indirizzi è suddiviso in pagine virtuali di uguale dimensione, alcune delle quali sono contenute in blocchi fisici di memoria. Un accesso a

una pagina in memoria è tradotto dalla MMU nell'indirizzo fisico corretto, altrimenti il tentativo di accesso causa un errore di pagina. Pentium 4 e UltraSPARC III dispongono entrambi di MMU che gestiscono la memoria virtuale e la paginazione.

L'astrazione di I/O più importante presente a questo livello è il file. Un file consiste in una sequenza di byte o di record logici che può essere letta o scritta senza alcuna nozione del funzionamento dei dischi, delle unità a nastro o di altri dispositivi di I/O. L'accesso ai file può essere sequenziale o diretto (per chiave o per numero di record). Le directory sono utilizzate per raggruppare file. Questi possono essere memorizzati in settori consecutivi del disco o sparpagliati al suo interno. Nel secondo caso, che è il più frequente nei dischi normali, si fa uso di strutture dati per la localizzazione dei blocchi di un file. È possibile mantenere traccia delle locazioni libere del disco tramite una lista o una bit map.

Il calcolo parallelo viene spesso gestito e implementato tramite la simulazione di processi multipli che condividono a tempo la singola CPU. Le interazioni tra processi, se non controllate opportunamente, possono portare a corse critiche. Per scongiurare questo problema si introducono primitive di sincronizzazione, di cui un esempio è dato dai semafori. Grazie ai semafori è possibile risolvere problemi analoghi a quello del produttore-consumatore in maniera semplice ed elegante.

XP e UNIX sono due esempi di sistemi operativi molto sofisticati. Entrambi supportano paginazione, corrispondenza tra file in memoria e file system gerarchico, ed entrambi intendono i file come semplici sequenze di byte. Infine tutti e due i sistemi operativi forniscono la gestione dei processi e dei thread, nonché diverse modalità per la loro sincronizzazione.

Problemi

1. Perché un sistema operativo interpreta solo alcune delle istruzioni del livello 3, mentre un microprogramma interpreta tutte le istruzioni del livello ISA?
2. Una macchina ha uno spazio degli indirizzi virtuali di 32 bit indirizzabile al byte. La dimensione di pagina è di 4 KB. Quante pagine d'indirizzi virtuali esistono?
3. È necessario che le pagine abbiano dimensioni pari a potenze di 2? Sarebbe possibile, in teoria, implementare pagine di 4000 byte? Se sì, si tratterebbe di una soluzione pratica?
4. Una memoria virtuale ha dimensioni di pagina di 1024 parole, otto pagine virtuali e quattro blocchi fisici di memoria. La tabella delle pagine è la seguente:

Pagina virtuale	Blocco di memoria
0	3
1	1
2	assente dalla memoria
3	assente dalla memoria
4	2
5	assente dalla memoria
6	0
7	assente dalla memoria

   a. Elencare tutti gli indirizzi virtuali che causerebbero un errore di pagina.
   b. Quali sono gli indirizzi fisici corrispondenti a 0, 3728, 1023, 1024, 1025, 7800 e 4096?
5. Un computer ha 16 pagine d'indirizzi virtuali e solo quattro blocchi di memoria. La memoria è inizialmente vuota. Se un programma accede alle pagine virtuali secondo l'ordine

   0, 7, 2, 7, 5, 8, 9, 2, 4

   a. Quali riferimenti causerebbero un errore di pagina se si usa LRU?
   b. Quali riferimenti causerebbero un errore di pagina se si usa FIFO?
6. Nel Paragrafo 6.1.4 è stato introdotto un algortimo per l'implementazione della sostituzione delle pagine secondo la strategia FIFO. Se ne progetti una versione più efficiente. *Suggerimento:* è possibile limitarsi ad aggiornare il contatore della sola pagina appena caricata, lasciando invariato quello delle altre pagine.
7. Nei sistemi paginati trattati nel capitolo, il gestore degli errori di pagina faceva parte del livello ISA e non era contenuto perciò nello spazio degli indirizzi di alcun programma del livello OSM. In realtà lo stesso gestore degli errori di pagina occupa alcune pagine e potrebbe venire esso stesso rimosso in determinate circostanze (per esempio se si adottasse la politica di sostituzione delle pagine FIFO). Che cosa accadrebbe se il gestore degli errori di pagina non si trovasse in memoria quando capita un errore di pagina? Come si potrebbe risolvere il problema?

8. Non tutti i computer dispongono di un meccanismo che asserisce automaticamente un bit hardware quando viene effettuata una scrittura in una pagina. Nondimeno è utile tener traccia delle pagine modificate al fine di evitare la scrittura su disco di tutte le pagine caricate (quando non servono più). Se in ogni pagina esistono alcuni bit hardware per l'abilitazione separata dei permessi in lettura, in scrittura e in esecuzione, come può il sistema operativo usare questi bit per stabilire se una pagina è "pulita" o se è stata modificata?

9. Una memoria segmentata è a paginazione dei segmenti. Ogni indirizzo virtuale ha 2 bit per il numero di segmento, 2 bit per il numero di pagina e 11 bit per l'offset all'interno della pagina. La memoria principale contiene 32 KB, suddivisi in pagine di 2 KB. Per ogni segmento sono possibili solo le configurazioni seguenti: in sola lettura, in lettura/esecuzione, in lettura/scrittura o in lettura/scrittura/esecuzione. La tabella delle pagine e le modalità di protezione sono le seguenti:

Segmento 0		Segmento 1		Segmento 2	Segmento 3	
Sola lettura		Lettura/esecuzione		Lettura/scrittura/esecuzione	Lettura/scrittura	
Pagina virtuale	Blocco di memoria	Pagina virtuale	Blocco di memoria		Pagina virtuale	Blocco di memoria
0	9	0	su disco	Tabella	0	14
1	3	1	0	delle pagine	1	1
2	su disco	2	15	non presente	2	6
3	12	3	8	in memoria principale	3	su disco

Per ognuno dei seguenti accessi alla memoria virtuale si indichi l'indirizzo fisico che viene calcolato. Se si verifica un errore di pagina, specificarne il tipo.

Accesso	*Segmento*	*Pagina*	*Offset nella pagina*
1. fetch di dati	0	1	1
2. fetch di dati	1	1	10
3. fetch di dati	3	3	2047
4. memorizzazione di dati	0	1	4
5. memorizzazione di dati	3	1	2
6. memorizzazione di dati	3	0	14
7. salto a	1	3	100
8. fetch di dati	0	2	50
9. fetch di dati	2	0	5
10. salto a	3	0	60

10. Alcuni calcolatori consentono l'I/O direttamente nello spazio dell'utente. Per esempio, un programma potrebbe avviare un trasferimento dal disco verso un buffer di un processo dell'utente. Questa possibilità può causare problemi se la memoria virtuale è implementata con compattamento? Si argomenti la risposta.

11. I sistemi operativi che permettono la corrispondenza tra file in memoria richiedono che essa avvenga su multipli di pagine (e che rispetti le estremità delle pagine). Per esempio, se le pagine sono di 4 KB, un file può essere mappato a partire dall'indirizzo virtuale 4096, ma non dall'indirizzo 5000. Perché?

12. Quando nel Pentium 4 viene caricato un registro di segmento, il descrittore corrispondente viene rintracciato e caricato all'interno di una porzione invisibile del registro di segmento. Si dia una giustificazione per questa scelta dei progettisti Intel.

13. Un programma del Pentium 4 effettua un accesso al segmento locale 10 con offset 8000. Il campo BASE dell'elemento di LDT corrispondente al segmento 10 contiene il numero 10000. Quale elemento della directory delle pagine usa il Pentium 4? Qual è il numero di pagina? Qual è l'offset?

14. Si confrontino alcuni possibili algoritmi per l'estromissione dei segmenti in una memoria segmentata, ma non paginata.

15. Si confrontino la frammentazione interna e quella esterna. Che cosa si può fare per ridurle?

16. I supermercati sono abituati ad affrontare un problema molto simile a quello della sostituzione di pagine nei sistemi con memoria virtuale. I loro scaffali hanno una capienza fissa a fronte di un numero di prodotti sempre crescente. Se viene commercializzato un nuovo prodotto rivoluzionario, diciamo un cibo per cani molto proteico, sarà necessario fargli spazio estromettendo alcuni prodotti dall'inventario. Gli algoritmi disponibili sono i soliti LRU e FIFO. Quale sarebbe preferibile in questo contesto?

17. La cache e la paginazione sono per molti versi simili. In entrambi i casi ci sono due livelli di memoria (la cache e la memoria principale nel primo caso, la memoria e il disco nel secondo). In questo capitolo abbiamo enunciato alcune argomentazioni a favore delle pagine di disco grandi e altre a favore di quelle piccole. Valgono le stesse argomentazioni per la dimensione delle linee di cache?

18. Perché molti file system richiedono l'apertura preventiva di un file con una chiamata di sistema `open` prima della lettura?

19. Si confrontino il metodo che usa la bit map e quello che usa la lista delle lacune per la gestione dello spazio libero su di un disco con 800 cilindri, ciascuno costituito di 5 tracce di 32 settori. Quante lacune ci vorrebbero perché la lista relativa ecceda la dimensione della bit map? Si ipotizzi che l'unità di allocazione sia il settore e che una lacuna richieda un elemento di tabella di 32 bit.

20. Al fine di prevedere le prestazioni del disco è utile avere un modello dell'allocazione dei dati. Si supponga che il disco sia uno spazio lineare degli indirizzi con svariati settori, costituito da una sequenza di blocchi di dati, quindi da una lacuna, da un'altra sequenza di blocchi di dati, e così via. Se alcune misure empiriche dimostrano che le distribuzioni di probabilità dei dati e delle lacune sono uguali e attribuiscono probabilità 2^{-i} alle sequenze di lunghezza i, qual è il valore atteso delle lacune in un disco?

21. In un certo calcolatore, un programma può creare tanti file quanti sono necessari, e tutti i suoi file possono crescere dinamicamente durante l'esecuzione, senza dare alcuna indicazione preventiva al sistema operativo circa la loro dimensione finale. Su un sistema siffatto i file verranno memorizzati in settori consecutivi? Si motivi la risposta.

22. Le statistiche hanno dimostrato che più della metà dei file ha dimensioni di pochi KB e che la stragrande maggioranza dei file è al di sotto di un certo numero di KB (attorno agli 8 KB). D'altro canto, il 10% dei file più voluminosi è responsabile del 95% dell'intera occupazione di spazio su disco. A partire da questi dati, che conclusione si può trarre circa la dimensione dei blocchi del disco?

23. Si consideri una possibile implementazione delle istruzioni sui semafori da parte di un sistema operativo: quando la CPU sta per effettuare una `up` o `down` su un semaforo (una variabile intera in memoria), il sistema imposta o maschera i bit di priorità della CPU in modo tale da disabilitare tutti gli interrupt. Quindi effettua il fetch del semaforo, lo modifica e salta in base al suo valore. Solo a questo punto riabilita gli interrupt. Il metodo funziona se:

a. c'è una sola CPU che effettua la commutazione tra processi ogni 100 ms

b. due CPU condividono una memoria comune in cui si trova il semaforo.

24. I produttori del Sistema Operativo Indistruttibile hanno ricevuto delle lamentele da parte di clienti a proposito dell'ultima distribuzione del sistema, in cui sono state introdotte le operazioni sui semafori. Secondo i clienti è immorale che un processo si blocchi (lo definiscono "poltrire"). Dato che l'azienda crede nel motto secondo cui il cliente ha sempre ragione, si è pensato di introdurre una terza operazione a supporto di `up` e `down`: l'operazione `peek` ("sbirciare"), che si limita a esaminare il valore di un semaforo senza modificarlo e senza sospendere l'esecuzione. In tal modo, se un programma crede sia immorale fermare la propria esecuzione, può cominciare con l'ispezionare il semaforo e stabilire se è sicuro effettuare una `down`. Questa soluzione funziona se il semaforo è usato da tre o più processi? E se è usato da due processi?

25. Redigere una tabella che indichi, in funzione dei tempi sotto indicati (da 0 a 1000 ms), quale dei processi P1, P2 e P3 sia in esecuzione e quale sia bloccato. Tutti e tre i processi effettuano istruzioni `up` e `down` sullo stesso semaforo. Quando due processi sono bloccati e viene eseguita una `up`, viene risvegliato il processo con nominativo più basso, ovvero P1 ha precedenza su P2 e P3, e così via. All'inizio tutti e tre i processi sono in esecuzione e il semaforo vale 1.

Al tempo t = 100 P1 effettua una `down`
Al tempo t = 200 P1 effettua una `down`
Al tempo t = 300 P2 effettua una `up`
Al tempo t = 400 P3 effettua una `down`
Al tempo t = 500 P1 effettua una `down`
Al tempo t = 600 P2 effettua una `up`
Al tempo t = 700 P2 effettua una `down`
Al tempo t = 800 P1 effettua una `up`
Al tempo t = 900 P1 effettua una `up`

26. Nei sistemi di prenotazioni aeree è necessario assicurare che durante l'accesso a un file da parte di un processo, nessun altro possa accedervi. In caso contrario due processi diversi, creati per conto di due compagnie aeree diverse, potrebbero prenotare inavvertitamente lo stesso posto dello stesso volo. Si progetti un metodo di sincronizzazione che usi i semafori per garantire l'accesso esclusivo dei processi ai file (ipotizzando che i processi obbediscano alle regole proposte).

27. Gli architetti degli elaboratori spesso forniscono un'istruzione `TSL` (*Test and Set Lock*, "esamina e riserva") per rendere possibile l'implementazione dei semafori sui computer con più CPU che condividono la stessa memoria. `TSL X` esamina la locazione X: se contiene il valore 0, viene posta a 1 all'interno di un solo, indivisibile ciclo di memoria, e l'istruzione successiva viene saltata. Se invece non vale 0, la `TSL` si comporta come una `NOP`. Grazie a `TSL` è possibile scrivere procedure *lock* e *unlock* con le seguenti proprietà: *lock(x)* verifica se x è riservata, in caso contrario la riserva e restituisce il controllo. Se x è riservata invece aspetta finché non venga rilasciata, dopo di che la riserva e restituisce il controllo. *unlock* libera una variabile riservata. Se tutti i processi riservano la tabella dei semafori prima di usarla, solo un processo per volta può manipolare le variabili e i puntatori, evitando così le corse critiche. Si scrivano le procedure *lock* e *unlock* in linguaggio assemblativo (dopo aver fatto le ipotesi ritenute ragionevoli).

28. Si mostrino i valori di *in* e *out* per un buffer circolare lungo 65 parole dopo ciascuna delle seguenti operazioni. Entrambi partono da 0.

a. Inserimento di 22 parole.
b. Estrazione di 9 parole.
c. Inserimento di 40 parole.

d. Estrazione di 17 parole.
e. Inserimento di 12 parole.
f. Estrazione di 45 parole.
g. Inserimento di 8 parole.
h. Estrazione di 11 parole.

29. Se una versione di UNIX usa blocchi del disco di 2 KB e memorizza in ogni blocco indiretto (singolo, doppio o triplo) 512 indirizzi del disco, qual è la dimensione massima di file? (si considerino puntatori a file di 64 bit).

30. Si supponga che la chiamata di sistema di UNIX

```
unlink( "/usr/ast/bin/gioco3")
```

sia eseguita nel contesto della Figura 6.36. Si descrivano con cura i cambiamenti risultanti nel directory system.

31. Si immagini di dover implementare UNIX su di un microcomputer con pochissima memoria. Se, nonostante tutti gli sforzi adoperati, il sistema risulta ancora troppo grande, si rende necessario il sacrificio di una chiamata di sistema. La scelta ricade su `pipe`, usata per la creazione delle pipe che permettono la comunicazione di flussi di byte tra processi. È ancora possibile implementare la redirezione di I/O in qualche modo? E le pipeline? Si analizzino il problema e le sue possibili soluzioni.

32. I membri della commissione per l'Uguaglianza dei Descrittori di File stanno organizzando una protesta contro UNIX perché le sue chiamate che restituiscono descrittori di file, restituiscono sempre il numero più piccolo non ancora in uso. Di conseguenza i descrittori di file corrispondenti ai numeri grandi non vengono quasi mai usati. La loro proposta è di far sì che venga restituito il numero più piccolo non ancora usato dal programma, invece che quello più piccolo attualmente inutilizzato. Sostengono che si tratta di una richiesta banale da implementare, che non comporterà la modifica dei programmi esistenti e che renderà il sistema più equo. Che cosa ne pensate?

33. In XP è possibile generare una lista di controllo degli accessi affinché Roberta non abbia alcun accesso a un certo file e tutti gli altri utenti ne abbiano pieno accesso. Quale può essere l'implementazione di una lista siffatta?

34. Si descrivano due modi diversi di programmare il problema del produttore-consumatore usando i buffer condivisi e i semafori di XP. Si rifletta in particolar modo sull'implementazione del buffer condiviso nelle due soluzioni proposte.

35. Capita frequentemente di usare la simulazione per valutare il comportamento degli algoritmi di sostituzione delle pagine. A tal fine, si scriva un simulatore per una memoria virtuale paginata, con 64 pagine di 1 KB. Il simulatore dovrebbe mantenere in memoria una tabella con 64 elementi, uno per pagina, per indicare il numero della pagina fisica corrispondente a ogni pagina virtuale. Il simulatore dovrebbe leggere da un file gli indirizzi virtuali scritti in decimale, uno per riga. Se la pagina corrispondente è in memoria, è sufficiente annotare un successo di pagina (un page hit). Se non è in memoria, il simulatore chiama una procedura per la scelta della pagina da estromettere dalla memoria (cioè per la scelta dell'elemento della tabella da sovrascrivere) e registra un fallimento di pagina (un page miss). Nella realtà non avviene alcuno spostamento di pagina, ma solo la sua simulazione. Si generi un file d'indirizzi casuali e si valutino le prestazioni di LRU e di FIFO. Quindi si generi un file d'indirizzi in cui una frazione costante x degli indirizzi sia ottenuta sommando quattro byte agli indirizzi appena precedenti (per simulare il principio di località). Si eseguano dei test per valori diversi di x e si riferiscano i risultati ottenuti.

36. Il programma della Figura 6.25 ha una corsa critica fatale perché due thread accedono alle stesse variabili condivise in modo non controllato, senza usare semafori o altre tecniche di mutua esclusione. Si esegua il programma e si osservi quanto tempo passa prima del suo bloccaggio. Se non si osserva alcun bloccaggio, si incrementino le dimensioni della finestra di vulnerabilità inserendo qualche istruzione tra la modifica di `m.in` e `m.out` e la loro valutazione. Quante istruzioni bisogna aggiungere perché l'esecuzione si fermi, per esempio, una volta all'ora?

37. Si scriva un programma per UNIX o per XP che prende in input il nome di una directory e che stampa la lista dei file in essa contenuti, uno per riga. Il nome di ogni file va fatto seguire dalla stampa della sua dimensione. Si stampino i nomi dei file nell'ordine in cui si trovano nella directory. Gli elementi inutilizzati di una directory devono essere stampati con la dicitura (inutilizzato).

Capitolo 7

Livello del linguaggio assemblativo

Mentre nei Capitoli 4, 5 e 6 abbiamo illustrato tre diversi livelli che si trovano nella maggior parte degli odierni calcolatori, qui affronteremo principalmente un ulteriore livello, presente anch'esso su praticamente tutti i calcolatori moderni, quello del linguaggio assemblativo. La differenza principale che contraddistingue questo livello rispetto ai livelli di microarchitettura, ISA e macchina del sistema operativo è che il livello del linguaggio assemblativo è implementato mediante traduzione invece che interpretazione.

I programmi che si occupano di tradurre in un particolare linguaggio un programma scritto dall'utente in un altro linguaggio sono chiamati **traduttori**. Il linguaggio nel quale è scritto il programma originale viene chiamato **linguaggio sorgente**, e **linguaggio destinazione** quello nel quale viene convertito. Sia il linguaggio sorgente sia il linguaggio destinazione definiscono dei livelli. Se esistesse un processore in grado di eseguire direttamente i programmi scritti nel linguaggio sorgente, tradurre il programma originale nel linguaggio destinazione non sarebbe necessario.

Si usa la traduzione quando si ha a disposizione un processore (un processore hardware oppure un interprete) per il linguaggio destinazione, ma non per il linguaggio sorgente. Se la traduzione viene effettuata correttamente, l'esecuzione del programma tradotto fornisce esattamente lo stesso risultato che si otterrebbe dal programma sorgente se si avesse a disposizione un processore in grado di eseguirlo. Di conseguenza è possibile implementare un nuovo livello per il quale non esiste un processore, traducendo inizialmente i suoi programmi in un livello destinazione ed eseguendo poi i programmi del livello destinazione ottenuti dalla traduzione.

È importante notare la differenza che esiste fra traduzione, da una parte, e interpretazione, dall'altra. Nella traduzione il programma originale non viene eseguito direttamente, ma viene invece convertito in un programma equivalente chiamato **programma oggetto** o **programma eseguibile binario** la cui esecuzione è portata avanti solo dopo che la traduzione è stata completata. La traduzione viene realizzata in due passi distinti:

1. generazione di un programma equivalente nel linguaggio destinazione
2. esecuzione del programma appena generato.

Questi due passi non sono simultanei: il secondo non inizia finché il primo non sia stato completato. Al contrario, nell'interpretazione esiste un unico passo: l'esecuzione del pro-

gramma sorgente originale. Non occorre generare, in una fase iniziale, alcun programma equivalente, anche se a volte il programma sorgente può essere convertito in una forma intermedia (per esempio il bytecode di Java) per facilitarne l'interpretazione.

Mentre si esegue il programma oggetto sono coinvolti solamente tre livelli: il livello di microarchitettura, il livello ISA e il livello macchina del sistema operativo. Di conseguenza, durante l'esecuzione è possibile trovare nella memoria del calcolatore tre diversi programmi: il programma oggetto dell'utente, il sistema operativo e il microprogramma (se esiste), mentre è svanita qualsiasi traccia del programma sorgente originale. Il numero di livelli presenti al momento dell'esecuzione può dunque essere diverso dal numero di livelli presenti prima di effettuare la traduzione. Occorre tuttavia tener presente che, mentre noi definiamo un livello in base alle istruzioni e ai costrutti linguistici resi disponibili ai suoi programmatori (e non solo in base alle tecniche implementative), altri autori distinguono nettamente i livelli implementati dagli interpreti che lavorano a tempo di esecuzione dai livelli implementati mediante la traduzione.

7.1 Introduzione al linguaggio assemblativo

A seconda della relazione che intercorre tra il linguaggio sorgente e il linguaggio destinazione è possibile dividere sommariamente i traduttori in due gruppi. Quando il linguaggio sorgente è essenzialmente una rappresentazione simbolica di un linguaggio macchina numerico, il traduttore è chiamato **assemblatore** e il linguaggio sorgente è chiamato **linguaggio assemblativo**. Quando il linguaggio sorgente è un linguaggio ad alto livello come Java oppure C e il linguaggio destinazione è un linguaggio macchina numerico oppure una sua rappresentazione simbolica: in questo caso il traduttore viene chiamato **compilatore**.

7.1.1 Che cos'è un linguaggio assemblativo

Un puro linguaggio assemblativo è un linguaggio nel quale ciascuna istruzione produce esattamente un'istruzione macchina. In altre parole esiste una corrispondenza uno-a-uno tra le istruzioni macchina e le istruzioni del programma assemblativo. Se ciascuna linea del linguaggio assemblativo contiene esattamente un'istruzione macchina, un programma assemblativo di n linee genererà un programma in linguaggio macchina di n parole.

La ragione dell'utilizzo del linguaggio assemblativo al posto del linguaggio macchina (in esadecimale) è che è molto più facile programmare utilizzando il linguaggio assemblativo. L'uso di una forma simbolica per i nomi e gli indirizzi, al posto della rappresentazione binaria oppure ottale, costituisce un'enorme differenza. La maggior parte dei programmatori è in grado di ricordare che `ADD`, `SUB`, `MUL` e `DIV` sono le abbreviazioni di aggiungi, sottrai, moltiplica e dividi, mentre pochi riescono a ricordarsi i corrispondenti valori numerici utilizzati dalla macchina. Il programmatore in linguaggio assemblativo deve ricordarsi solamente i nomi simbolici, dato che sarà compito dell'assemblatore tradurli in istruzioni macchina.

La stessa osservazione vale per gli indirizzi. Un programmatore può assegnare nomi simbolici alle locazioni di memoria e lasciare all'assemblatore il compito di determinare i corrispondenti valori numerici. Programmando direttamente in linguaggio macchina è invece obbligatorio lavorare sempre con i valori numerici degli indirizzi. Per questi motivi al giorno d'oggi nessuno programma più in linguaggio macchina, anche se alcuni decenni fa, prima che venissero inventati gli assemblatori, era necessario farlo.

Il linguaggio assemblativo, oltre alla corrispondenza uno-a-uno tra le sue istruzioni e le istruzioni macchina, gode di un'altra proprietà che lo contraddistingue dai linguaggi ad alto livello. Il programmatore in linguaggio assemblativo ha accesso a tutte le funzionalità e a tutte le istruzioni disponibili nella macchina di destinazione, mentre il programmatore in linguaggio ad alto livello non ha la stessa libertà di accesso. Se per esempio la macchina di destinazione ha un bit di overflow, un programma assemblativo può testarlo, mentre un programma Java non lo può fare direttamente. Un programma assemblativo, diversamente da uno ad alto livello, può quindi eseguire qualsiasi istruzione dell'insieme delle istruzioni della macchina di destinazione. In poche parole, tutto ciò che può essere fatto in linguaggio macchina, può anche essere fatto in linguaggio assemblativo. Al contrario, un programmatore in linguaggio ad alto livello non ha a disposizione tutte le istruzioni, i registri e le funzionalità della macchina. I linguaggi per la programmazione di sistemi, come C, sono spesso trasversali rispetto a questi due tipi, dato che la loro sintassi è ad alto livello, ma forniscono allo stesso tempo molte delle possibilità di accesso alla macchina tipiche dei linguaggi assemblativi.

Un'ultima differenza che vale la pena rendere più esplicita è che i programmi in linguaggio assemblativo vengono scritti per una specifica famiglia di macchine, mentre un programma scritto in un linguaggio ad alto livello può essere eseguito su molte macchine diverse. La possibilità di portare il software da una macchina a un'altra rappresenta un grande vantaggio per molte applicazioni.

7.1.2 Perché usare il linguaggio assemblativo

Programmare in linguaggio assemblativo è difficile. Non ci si faccia illusioni. Non è un lavoro per programmatori pavidi e debolucci. Inoltre scrivere un programma in linguaggio assemblativo richiede molto più tempo che scriverlo in un linguaggio ad alto livello, e anche la correzione degli errori e l'aggiornamento del codice sono operazioni molto più complesse.

Ma allora, perché mai si dovrebbe programmare in linguaggio assemblativo? Esistono due motivi: le prestazioni e le possibilità di accesso alla macchina. Prima di tutto un esperto programmatore in linguaggio assemblativo può spesso creare codice molto più piccolo e veloce di quanto possa fare un programmatore in linguaggio ad alto livello. Per alcune applicazioni la velocità e la dimensione rappresentano due fattori critici; in questa categoria ricadono i driver dei dispositivi, le routine del BIOS, i cicli interni di programmi le cui prestazioni hanno un'importanza fondamentale e molte delle applicazioni integrate, come il codice di una smart card o di un telefono cellulare.

In secondo luogo, alcune procedure devono accedere in modo completo all'hardware (cosa che in genere è impossibile con i linguaggi ad alto livello). In questa categoria ritroviamo per esempio i gestori a basso livello degli interrupt e delle eccezioni dei sistemi operativi e i controllori dei dispositivi di molti sistemi integrati che funzionano in tempo reale.

Il primo motivo che spinge a programmare in linguaggio assemblativo (l'ottenimento di prestazioni più elevate) è di solito quello più importante; analizziamolo quindi più da vicino. Nella maggior parte dei programmi una piccola percentuale dell'intero codice è responsabile di un'elevata percentuale del tempo di esecuzione. Capita comunemente che l'1%

del programma sia responsabile del 50% del tempo di esecuzione e che il 10% del programma ne sia responsabile per il 90%.

Assumiamo, per esempio, che scrivere un programma in un linguaggio ad alto livello richieda 10 anni-uomo di lavoro e che il programma risultante impieghi 100 s per eseguire un determinato *benchmark* (cioè un programma di test impiegato per confrontare le prestazioni di calcolatori, compilatori, ecc.). Scrivere l'intero programma in linguaggio assemblativo richiederebbe invece 50 anni-uomo di lavoro, dato che i programmatori in linguaggio assemblativo sono meno produttivi. Il programma risultante potrebbe eseguire il benchmark in soli 33 s, in quanto un abile programmatore può battere un efficiente compilatore di un fattore 3 (anche se si potrebbe disquisire senza fine a proposito di questo rapporto). La situazione è mostrata nella Figura 7.1.

Basandosi sulla precedente osservazione, secondo la quale solo una piccola frazione del codice è responsabile della maggior parte del tempo di esecuzione, è possibile seguire un approccio diverso. Il programma viene inizialmente scritto in un linguaggio ad alto livello e successivamente si esegue una serie di misurazioni per stabilire quali parti del programma determinano la maggior parte del tempo di esecuzione. Tra queste misurazioni troviamo generalmente l'uso del clock di sistema per calcolare la quantità di tempo spesa in ciascuna procedura, e il conteggio del numero di volte che ciascun ciclo viene eseguito.

Come esempio assumiamo che il 10% dell'intero programma sia responsabile del 90% del tempo di esecuzione. È possibile migliorare il 10% più critico riscrivendolo in linguaggio assemblativo. Questo processo è chiamato **messa a punto** ed è illustrato nella Figura 7.1. In questo caso la riscrittura delle procedure richiede ulteriori 5 anni-uomo di programmazione, ma, allo stesso tempo, permette di ridurre il tempo di esecuzione da 90 a 30 s.

Può essere interessante confrontare l'approccio che unisce il linguaggio ad alto livello e il linguaggio assemblativo con la versione del programma che impiega solamente il linguaggio assemblativo (vedi Figura 7.1). Il programma ottenuto con la seconda tecnica risulta più veloce del 20% (33 s contro 40), ma questo miglioramento è ottenuto a un prezzo tre volte maggiore (50 anni-uomo di programmazione contro 15). Inoltre il vantaggio reale è ancora maggiore, dato che riprogrammare in codice assembaltivo una procedura già testata e corretta in linguaggio ad alto livello è in realtà molto più semplice che scriverla da zero in linguaggio assemblativo. In altre parole la stima di 5 anni-uomo di programmazione necessari per riscrivere le procedure più critiche è sovrabbondante. Se la riscrittura del codice richiedesse solamente 1 anno-uomo di lavoro, il rapporto tra il costo dell'approccio misto e il costo dell'approccio puramente assemblativo sarebbe superiore di 4 a 1, a favore dell'approccio misto.

	Anni/uomo di programmazione richiesti	Tempo d'esecuzione del programma in secondi
Linguaggio assemblativo	50	33
Linguaggio ad alto livello	10	100
Approccio misto prima della messa a punto		
10% più critico	1	90
Restante 90%	9	10
Totale	10	100
Approccio misto dopo la messa a punto		
10% più critico	6	30
Restante 90%	9	10
Totale	15	40

Figura 7.1 Confronto tra linguaggio assemblativo e linguaggio ad alto livello, con e senza messa a punto.

Utilizzando un linguaggio ad alto livello non ci si deve dedicare allo spostamento di bit da una parte all'altra e spesso ciò permette di avere una comprensione più profonda del problema, consentendo in alcuni casi di migliorare *realmente* le prestazioni. Questa situazione si verifica molto più raramente ai programmatori di linguaggio assemblativo, che in genere si destreggiano con le istruzioni per risparmiare anche pochi cicli.

In conclusione esistono almeno quattro buone ragioni per studiare il linguaggio assemblativo. In primo luogo, il successo o il fallimento di un progetto di grandi dimensioni possono dipendere dalla capacità di spremere le prestazioni fino a ottenere, per alcune procedure critiche, un fattore di miglioramento di 2 oppure 3, e per ottenere ciò è importante essere in grado di scrivere un codice assemblativo di buon livello quando è realmente necessario farlo.

Inoltre, a volte il codice assemblativo è l'unica soluzione alla mancanza di memoria. Le smart card contengono una CPU, ma poche hanno un megabyte di memoria e ancora meno sono dotate di un hard disk per effettuare la paginazione. Ciononostante esse devono eseguire complessi calcoli crittografici utilizzando risorse limitate. Per ragioni di costo spesso i processori integrati negli elettrodomestici hanno una quantità minima di memoria. Per limitare il consumo energetico, i PDA e altri dispositivi elettronici senza fili alimentati a batterie sono spesso dotati di una memoria talmente piccola che è indispensabile scrivere codice molto efficiente.

La terza ragione è che un compilatore deve produrre un risultato utilizzabile da un assemblatore oppure eseguire esso stesso il processo di assemblaggio. Comprendere il linguaggio assemblativo è quindi essenziale per capire come lavorano i compilatori.

Infine, lo studio del linguaggio assemblativo permette di avere una visione più chiara del funzionamento delle macchine reali. Per gli studenti di architettura degli elaboratori scrivere anche poche righe di codice assemblativo è l'unico modo per avere consapevolezza di che cosa sono realmente le macchine a livello di microarchitettura.

7.1.3 Formato delle istruzioni del linguaggio assemblativo

La struttura delle istruzioni di un linguaggio assemblativo rispecchia la struttura delle istruzioni macchina, delle quali le prime forniscono una rappresentazione simbolica; ciononostante i linguaggi assemblativi di macchine diverse e di livelli differenti hanno un numero sufficiente di somiglianze da permettere di parlare di linguaggio assemblativo in termini generali. La Figura 7.2 mostra alcuni frammenti del linguaggio assemblativo del Pentium 4, del Motorola 680x0 e dell'(Ultra)SPARC, che eseguono l'istruzione $N = I + J$. Negli esempi le istruzioni sopra la linea vuota eseguono la computazione e quelle al di sotto sono invece i comandi che indicano all'assemblatore di riservare spazio di memoria per le variabili I, J e N e non rappresentano alcuna istruzione macchina.

Esistono differenti assemblatori per la famiglia Intel, ciascuno dei quali ha una propria sintassi. Nel corso di questo capitolo utilizzeremo per i nostri esempi il linguaggio as-

Etichetta	Codice operativo	Operandi	Commenti
FORMULA:	MOV	EAX,I	; registro EAX = I
	ADD	EAX,J	; registro EAX = I + J
	MOV	N,EAX	; N = I + J
I	DD	3	; riserva 4 byte inizializzati a 3
J	DD	4	; riserva 4 byte inizializzati a 4
N	DD	0	; riserva 4 byte inizializzati a 0

(a)

Etichetta	Codice operativo	Operandi	Commenti
FORMULA	MOVE.L	I, D0	; registro D0 = I
	ADD.L	J, D0	; registro D0 = I + J
	MOVE.L	D0, N	; N = I + J
I	DC.L	3	; riserva 4 byte inizializzati a 3
J	DC.L	4	; riserva 4 byte inizializzati a 4
N	DC.L	0	; riserva 4 byte inizializzati a 0

(b)

Etichetta	Codice operativo	Operandi	Commenti
FORMULA:	SETHI	%HI(I),%R1	! R1 = bit più significativi dell'indirizzo di I
	LD	[%R1 + %LO(I)],%R1	! R1 = I
	SETHI	%HI(J),%R2	! R2 = bit più significativi dell'indirizzo di J
	LD	[%R2 + %LO(J)],%R2	! R2 = J
	NOP		! attende J dalla memoria
	ADD	%R1,%R2,%R2	! R2 = R1 + R2
	SETHI	%HI(N),%R1	! R1 = bit più significativi dell'indirizzo di N
	ST	%R2,[%R1 + %LO(N)]	
I:	.WORD 3		! riserva 4 byte inizializzati a 3
J:	.WORD 4		! riserva 4 byte inizializzati a 4
N:	.WORD 0		! riserva 4 byte inizializzati a 0

(c)

Figura 7.2 Esecuzione di $N = I + J$. (a) Pentium 4. (b) Motorola 680x0. (c) SPARC.

semblativo Microsoft MASM. Anche se ci concentreremo sul Pentium 4, tutto ciò che diremo varrà in modo equivalente anche per il 386, il 486, il Pentium e il Pentium Pro. Per gli esempi dello SPARC ci baseremo sull'assemblatore fornito dalla Sun. Anche in questo caso ogni cosa che diremo potrà essere applicata alle versioni precedenti (a 32 bit) di SPARC. Per ragioni di uniformità utilizzeremo sempre lettere maiuscole per i codici operativi e per i registri (seguendo la convenzione del Pentium 4), anche se l'assemblatore della Sun richiede lettere minuscole.

Le istruzioni del linguaggio assemblativo sono composte da quattro parti: un campo etichetta, un campo per l'operazione (codice operativo o *opcode*), un campo per gli operandi e un campo per i commenti. Le etichette sono utilizzate per fornire nomi simbolici agli indirizzi, e sono necessarie per definire le destinazioni alle quali portano le istruzioni che effettuano salti e per poter accedere alle parole di dati memorizzate mediante nomi simbolici. Se un'istruzione è etichettata, l'etichetta inizia (di solito) nella colonna 1.

Le tre sezioni della Figura 7.2 hanno quattro etichette: *FORMULA*, *I*, *J* e *N*. Si noti che il linguaggio assemblativo di SPARC richiede che ogni etichetta termini con il carattere

"due punti", mentre ciò non avviene per quello di Motorola. Il linguaggio assemblativo di Intel richiede i "due punti" solo nelle etichette del codice, ma non in quelle dei dati. Non c'è niente di importante in questa differenza: è solo che non tutti i progettisti di assemblatori hanno gli stessi gusti e non c'è alcun aspetto dell'architettura sottostante che suggerisca una scelta piuttosto che un'altra. Un vantaggio offerto dalla notazione che usa i "due punti" è che un'etichetta può comparire, da sola, su una linea, mentre il codice operativo viene scritto nella prima colonna della linea successiva. Questo stile si rivela in un certo modo utile per i compilatori. Senza i "due punti" sarebbe impossibile distinguere un'etichetta da un codice operativo, nel caso in cui entrambi occupino, da soli, una linea del codice assemblativo. Usando i "due punti" si elimina questa potenziale ambiguità.

Una caratteristica infelice di alcuni assemblatori limita a sei o a otto il numero di caratteri di un'etichetta, mentre la maggior parte dei linguaggi ad alto livello permette di utilizzare nomi di lunghezza arbitraria. Nomi lunghi e scelti in modo appropriato rendono i programmi molto più leggibili e facili da capire da chi non ne è l'autore.

Le tre le macchine sono dotate di registri con nomi molto differenti. I registri del Pentium 4 hanno nomi come `EAX`, `EBX`, `ECX`, ecc. I registri di Motorola sono chiamati `D0`, `D1`, `D2`, ecc. I registri SPARC hanno invece più nomi; nell'esempio abbiamo utilizzato i nomi `%R1` e `%R2`.

Il campo Codice operativo contiene un'abbreviazione simbolica del codice operativo oppure un comando per l'assemblatore stesso. La scelta di un nome appropriato è solo una questione di gusti e spesso i progettisti di differenti linguaggi assemblativi compiono scelte diverse. I progettisti dell'assemblatore Intel hanno deciso di usare `MOV` sia per caricare un registro dalla memoria sia per memorizzare un registro. I progettisti Motorola hanno scelto `MOVE` per entrambe le operazioni. Al contrario i progettisti dell'assemblatore SPARC hanno deciso di usare `LD` per la prima operazione e `ST` per la seconda. Anche queste scelte non hanno niente a che fare con la natura della macchina.

Contrariamente a quanto abbiamo appena visto, nel linguaggio assemblativo di SPARC, la necessità di utilizzare due istruzioni macchina, a partire da `SETHI`, per accedere alla memoria è una proprietà intrinseca della macchina. In SPARC infatti gli indirizzi virtuali possono essere a 32 bit (SPARC Version 8) oppure a 44 bit (SPARC Version 9), mentre le istruzioni possono gestire un massimo di 22 bit di dati immediati. Per fornire tutti i bit di un indirizzo virtuale completo sono quindi necessarie, sempre, due istruzioni. L'istruzione

```
SETHI %HI(I),%R1
```

azzera i 32 bit più significativi e i 10 meno significativi del registro (a 64 bit) `R1`, e successivamente inserisce i 22 bit dell'indirizzo a 32 bit di *I* nei bit compresi tra le posizioni 10 e 32 di `R1`. L'istruzione successiva

```
LD[%R1+%LO(I)],%R1
```

aggiunge `R1` e i 10 bit meno significativi dell'indirizzo di *I* per formare l'indirizzo completo di *I*, prelevare quella parola dalla memoria e inserirla in `R1`. In un concorso di bellezza con voti da 1 a 10, queste istruzioni otterrebbero un risultato di –20. SPARC non è però stato progettato per la bellezza del suo linguaggio assemblativo, ma per garantire un'alta velocità di esecuzione, e questo obiettivo riesce a raggiungerlo egregiamente.

Gli operandi della famiglia Pentium, del 680x0 e dello SPARC possono essere byte, parole e parole lunghe (4 byte): come fa l'assemblatore a sapere quale lunghezza è stata uti-

lizzata per l'operando? Anche in questo caso le scelte dei progettisti hanno portato a soluzioni differenti. Sul Pentium 4 i registri con lunghezze diverse hanno nomi differenti; EAX è utilizzato per esempio per spostare elementi di 32 bit, AX è utilizzato per spostare elementi di 16 bit, mentre AL e AH sono usati per spostare elementi di 8 bit. I progettisti dell'assemblatore Motorola hanno invece deciso di aggiungere a ciascun codice operativo il suffisso *.L* per le parole lunghe, *.W* per le parole e *.B* per i byte, invece di assegnare nomi diversi per i sottoinsiemi di D0, ecc. Al contrario SPARC usa codici operativi diversi per lunghezze differenti (per esempio, LDSB, LDSH e LDSW per caricare rispettivamente byte con segno, semiparole e parole in registri a 64 bit). Tutte e tre le soluzioni sono valide, ma sottolineano ancora una volta la natura arbitraria della progettazione di un linguaggio.

I tre assemblatori differiscono anche nel modo in cui riservano lo spazio per i dati. I progettisti del linguaggio assemblativo di Intel hanno scelto DD (*Define Double*), dato che sull'8088 una parola era di 16 bit. Quelli di Motorola hanno preferito DC (*Define Constant*), mentre quelli di SPARC hanno scelto fin dall'inizio .WORD. Ancora una volta si tratta di scelte arbitrarie.

Nelle istruzioni del linguaggio assemblativo il campo Operandi viene utilizzato per specificare gli indirizzi e i registri utilizzati come operandi dall'istruzione macchina. Nel caso di un'istruzione per la somma di interi il campo operandi indica quali elementi devono essere sommati fra loro. Nel caso di un'istruzione di salto indica invece a quale indirizzo deve portare la diramazione. Gli operandi possono essere registri, costanti, locazioni di memoria, e così via.

Il campo Commenti è il luogo in cui i programmatori possono inserire utili spiegazioni sul funzionamento del programma a beneficio di altri programmatori che in seguito lo potrebbero utilizzare o modificare (o a beneficio del programmatore stesso alcuni anni dopo aver scritto il codice). Un programma in linguaggio assemblativo senza una tale documentazione è praticamente incomprensibile, spesso anche per il suo stesso autore. Il campo dei commenti è destinato esclusivamente all'uso da parte dell'uomo e non ha alcun effetto sul processo di assemblaggio né tantomeno sul programma che verrà generato.

7.1.4 Pseudoistruzioni

Un linguaggio assemblativo, oltre a specificare quali istruzioni macchina devono essere eseguite, può anche contenere dei comandi indirizzati all'assemblatore stesso, per richiedere, per esempio, di allocare una certa quantità di memoria oppure per passare a una nuova pagina del listato. I comandi diretti all'assemblatore sono chiamati **pseudoistruzioni** o anche **direttive dell'assemblatore**. Nella Figura 7.2(a) abbiamo già visto una pseudoistruzione molto comune: DD (nella Figura 7.3 ne sono elencate altre, tratte dall'assemblatore Microsoft MASM per le macchine Intel).

La pseudoistruzione SEGMENT inizia un nuovo segmento, mentre ENDS lo termina. È consentito iniziare un segmento di testo, contenente codice, poi un segmento dati e tornare successivamente al segmento testo, e così via.

ALIGN forza l'indirizzo della linea successiva, contenente in genere dati, a essere un multiplo dell'argomento della pseudoistruzione. Per esempio se il segmento contiene già 61 byte di dati, l'indirizzo allocato subito dopo la pseudoistruzione ALIGN 4 sarà 64.

EQU assegna un nome simbolico a un'espressione. Per esempio, dopo la pseudoistruzione

```
BASE EQU 1000
```


Pseudoistruzione	Significato
SEGMENT	Inizia un nuovo segmento (testo, dati, ecc.) con certi attributi
ENDS	Termina il segmento corrente
ALIGN	Controlla l'allineamento della successiva istruzione o dei successivi dati
EQU	Definisce un nuovo simbolo uguale a una data espressione
DB	Alloca spazio per memorizzare uno o più byte (inizializzati)
DW	Alloca spazio per memorizzare uno o più elementi di dati (inizializzati) a 16 bit (parola)
DD	Alloca spazio per memorizzare uno o più elementi di dati (inizializzati) a 32 bit (parola doppia)
DQ	Alloca spazio per memorizzare uno o più elementi di dati (inizializzati) a 64 bit (parola quadrupla)
PROC	Inizia una procedura
ENDP	Termina una procedura
MACRO	Inizia una definizione di macro
ENDM	Termina una definizione di macro
PUBLIC	Esporta un nome definito nel modulo
EXTERN	Importa un nome da un altro modulo
INCLUDE	Preleva e include un altro file
IF	Inizia un assemblaggio condizionale in base a una data espressione
ELSE	Inizia un assemblaggio condizionale se la condizione IF precedente è falsa
ENDIF	Termina un assemblaggio condizionale
COMMENT	Definisce un nuovo carattere di inizio commento
PAGE	Genera un'interruzione di pagina nel listato
END	Termina il programma assemblativo

Figura 7.3 Pseudoistruzioni (direttive) dell'assemblatore del Pentium 4 (MASM).

il simbolo BASE può essere usato in qualsiasi punto al posto di 1000. L'espressione che segue EQU può comprendere alcuni simboli definiti dal programmatore oltre a operatori aritmetici e di altro tipo, come mostra il seguente esempio

```
LIMIT EQU 4 * BASE + 2000
```

La maggior parte degli assemblatori, compreso MASM, richiede che un simbolo sia definito prima di essere utilizzato in un'espressione come quella appena vista.

Le successive quattro pseudoistruzioni, DB, DW, DD e DQ, allocano memoria rispettivamente per una o più variabili della dimensione di 1, 2, 4 oppure 8 byte. Per esempio,

```
TABLE DB 11, 23, 49
```

alloca spazio per 3 byte e li inizializza rispettivamente ai valori 11, 23 e 49. Essa definisce inoltre il simbolo *TABLE* e lo associa all'indirizzo in cui è memorizzato il valore 11.

Le pseudoistruzioni PROC e ENDP definiscono rispettivamente l'inizio e la fine delle procedure del linguaggio assemblativo, che hanno la stessa funzione delle procedure definite

in altri linguaggi di programmazione. Analogamente le direttive MACRO e ENDM delimitano l'area in cui compare la definizione di una macroistruzione (argomento trattato più avanti nel corso del capitolo).

Le due successive pseudoistruzioni, PUBLIC e EXTERN, controllano la visibilità dei simboli. È una pratica comune scrivere i programmi in più file e spesso capita che una procedura contenuta in un file richiami una procedura o acceda ai dati definiti in un altro file. La pseudoistruzione PUBLIC permette di esportare un simbolo che deve essere reso disponibile anche ad altri file, rendendo così possibili i riferimenti tra file diversi. Analogamente, per impedire che l'assemblatore segnali un errore quando incontra un simbolo che non è ancora stato definito, si può dichiarare un simbolo come EXTERN. Questa pseudoistruzione indica all'assemblatore che la definzione del simbolo compare in un altro file. I simboli che non sono dichiarati mediante alcuna delle due pseudoistruzioni precedenti sono visibili soltanto localmente, all'interno dello stesso file. Ciò significa che l'utilizzo, per esempio, di *FOO* in più file non genererà alcun conflitto, dato che ciascuna definizione è locale al file in cui si trova.

La direttiva INCLUDE fa sì che l'assemblatore prelevi un altro file e lo includa interamente all'interno di quello corrente. Spesso tali file comprendono definizioni, macroistruzioni e altri elementi che sono richiesti in più file.

Molti assemblatori, compreso MASM, supportano l'assemblaggio condizionale. Per esempio,

```
WORDSIZE EQU 16
IF WORDSIZE GT 16
WSIZE:  DD 32
ELSE
WSIZE:  DD 16
ENDIF
```

alloca una singola parola a 32 bit e chiama *WSIZE* il suo indirizzo. La parola è inizializzata a 32 oppure a 16, in base al valore di *WORDSIZE* (in questo caso, 16). In genere si utilizza questo metodo per allocare una variabile quando occorre scrivere un programma che possa essere eseguito da macchine a 16 bit (come l'8088) o a 32 bit (come il Pentium 4). Inserendo all'interno di IF e ENDIF tutte le porzioni del codice dipendenti dalla macchina, e cambiando successivamente una singola definizione, *WORDSIZE*, il programma può essere automaticamente assemblato per entrambe le dimensioni di parola. Grazie a questo approccio è possibile mantenere un solo programma sorgente per più macchine destinazione (di tipo diverso), rendendo di conseguenza più facile lo sviluppo e il mantenimento del software. In molti casi tutte le definizioni dipendenti dalla macchina, come *WORDSIZE*, sono raggruppate in un singolo file, di cui vengono create versioni diverse per ogni tipo di macchina. In questo modo, includendo semplicemente il corretto file delle definizioni, è possibile utilizzare facilmente il programma per macchine diverse.

La pseudoistruzione COMMENT permette all'utente di modificare il delimitatore dei commenti, che inizialmente è impostato al carattere "punto e virgola". PAGE è utilizzato per controllare il listato che l'assemblatore può generare su richiesta. Infine END segna la fine del programma. Oltre a quelle appena elencate, in MASM esistono molte altre pseudoistruzioni. Altri assemblatori per il Pentium 4 hanno invece un insieme diverso di pseudoistruzioni; difatti queste non sono dettate dall'architettura della macchina, ma dai gusti di chi ha scritto l'assemblatore.

7.2 Macroistruzioni

Spesso i programmatori di linguaggi assemblativi hanno bisogno di ripetere più volte all'interno di un programma alcune sequenze d'istruzioni. Il modo più ovvio per farlo consiste nel riscriverle ogni volta che sia necessario. Tuttavia se una sequenza è lunga, o se deve essere utilizzata molte volte, scriverla ripetutamente è un'operazione tediosa.

Un approccio alternativo consiste nel trasformare la sequenza in una procedura e nel richiamarla quando è richiesta. Questa strategia ha lo svantaggio di dover eseguire, ogni volta che occorre utilizzare la sequenza, un'istruzione di chiamata di procedura e una per restituirne il controllo. Se le sequenze sono corte, per esempio di due istruzioni, ma sono usate frequentemente, il lavoro aggiuntivo determinato dalla chiamata alla procedura può rallentare notevolmente il programma. Le *macro*, abbreviazione usuale di macroistruzioni, forniscono una soluzione facile ed efficiente al problema di ripetere un'identica, o quasi, sequenza d'istruzioni.

7.2.1 Definizione, chiamata ed espansione di macro

Una **definizione di macro** è un modo per assegnare un nome a una porzione di testo. Dopo che una macro è stata definita, il programmatore può scriverne il nome al posto della porzione di programma corrispondente. Una macro in realtà non è altro che un'abbreviazione per una porzione di testo. La Figura 7.4(a) mostra un programma in linguaggio assemblativo per il Pentium 4 che scambia due volte il contenuto delle variabili *p* e *q*. Queste sequenze possono essere definite come macro, come mostra la Figura 7.4(b). Dopo aver definito la macro, ogni occorrenza di *SWAP* viene sostituita dalle quattro linee:

```
MOV EAX,P
MOV EBX,Q
MOV Q,EAX
MOV P,EBX
```

Il programmatore ha definito *SWAP* come un'abbreviazione delle quattro istruzioni.

Anche se assemblatori diversi utilizzano notazioni leggermente differenti, in tutti i casi una definizione di macro è composta dalle seguenti parti principali:

1. un'intestazione che indica il nome della macro che si sta definendo
2. il testo che costituisce il corpo della macro
3. una pseudoistruzione che segna la fine della definizione (per esempio, ENDM).

Quando l'assemblatore incontra una definizione di macro, la salva nella tabella delle definizioni di macro per poterla utilizzare successivamente. Da quel momento in poi ogni volta che il nome della macro appare come codice operativo, l'assemblatore la sostituisce con il corpo della macro. L'uso del nome di una macro come codice operativo viene detto **chiamata di macro**, mentre la sua sostituzione con il corpo della macro è chiamata **espansione di macro**.

in altri linguaggi di programmazione. Analogamente le direttive MACRO e ENDM delimitano l'area in cui compare la definizione di una macroistruzione (argomento trattato più avanti nel corso del capitolo).

Le due successive pseudoistruzioni, PUBLIC e EXTERN, controllano la visibilità dei simboli. È una pratica comune scrivere i programmi in più file e spesso capita che una procedura contenuta in un file richiami una procedura o acceda ai dati definiti in un altro file. La pseudoistruzione PUBLIC permette di esportare un simbolo che deve essere reso disponibile anche ad altri file, rendendo così possibili i riferimenti tra file diversi. Analogamente, per impedire che l'assemblatore segnali un errore quando incontra un simbolo che non è ancora stato definito, si può dichiarare un simbolo come EXTERN. Questa pseudoistruzione indica all'assemblatore che la definzione del simbolo compare in un altro file. I simboli che non sono dichiarati mediante alcuna delle due pseudoistruzioni precedenti sono visibili soltanto localmente, all'interno dello stesso file. Ciò significa che l'utilizzo, per esempio, di *FOO* in più file non genererà alcun conflitto, dato che ciascuna definizione è locale al file in cui si trova.

La direttiva INCLUDE fa sì che l'assemblatore prelevi un altro file e lo includa interamente all'interno di quello corrente. Spesso tali file comprendono definizioni, macroistruzioni e altri elementi che sono richiesti in più file.

Molti assemblatori, compreso MASM, supportano l'assemblaggio condizionale. Per esempio,

```
WORDSIZE EQU 16
IF WORDSIZE GT 16
WSIZE:  DD 32
ELSE
WSIZE:  DD 16
ENDIF
```

alloca una singola parola a 32 bit e chiama *WSIZE* il suo indirizzo. La parola è inizializzata a 32 oppure a 16, in base al valore di *WORDSIZE* (in questo caso, 16). In genere si utilizza questo metodo per allocare una variabile quando occorre scrivere un programma che possa essere eseguito da macchine a 16 bit (come l'8088) o a 32 bit (come il Pentium 4). Inserendo all'interno di IF e ENDIF tutte le porzioni del codice dipendenti dalla macchina, e cambiando successivamente una singola definizione, *WORDSIZE*, il programma può essere automaticamente assemblato per entrambe le dimensioni di parola. Grazie a questo approccio è possibile mantenere un solo programma sorgente per più macchine destinazione (di tipo diverso), rendendo di conseguenza più facile lo sviluppo e il mantenimento del software. In molti casi tutte le definizioni dipendenti dalla macchina, come *WORDSIZE*, sono raggruppate in un singolo file, di cui vengono create versioni diverse per ogni tipo di macchina. In questo modo, includendo semplicemente il corretto file delle definizioni, è possibile utilizzare facilmente il programma per macchine diverse.

La pseudoistruzione COMMENT permette all'utente di modificare il delimitatore dei commenti, che inizialmente è impostato al carattere "punto e virgola". PAGE è utilizzato per controllare il listato che l'assemblatore può generare su richiesta. Infine END segna la fine del programma. Oltre a quelle appena elencate, in MASM esistono molte altre pseudoistruzioni. Altri assemblatori per il Pentium 4 hanno invece un insieme diverso di pseudoistruzioni; difatti queste non sono dettate dall'architettura della macchina, ma dai gusti di chi ha scritto l'assemblatore.

7.2 Macroistruzioni

Spesso i programmatori di linguaggi assemblativi hanno bisogno di ripetere più volte all'interno di un programma alcune sequenze d'istruzioni. Il modo più ovvio per farlo consiste nel riscriverle ogni volta che sia necessario. Tuttavia se una sequenza è lunga, o se deve essere utilizzata molte volte, scriverla ripetutamente è un'operazione tediosa.

Un approccio alternativo consiste nel trasformare la sequenza in una procedura e nel richiamarla quando è richiesta. Questa strategia ha lo svantaggio di dover eseguire, ogni volta che occorre utilizzare la sequenza, un'istruzione di chiamata di procedura e una per restituirne il controllo. Se le sequenze sono corte, per esempio di due istruzioni, ma sono usate frequentemente, il lavoro aggiuntivo determinato dalla chiamata alla procedura può rallentare notevolmente il programma. Le *macro*, abbreviazione usuale di macroistruzioni, forniscono una soluzione facile ed efficiente al problema di ripetere un'identica, o quasi, sequenza d'istruzioni.

7.2.1 Definizione, chiamata ed espansione di macro

Una **definizione di macro** è un modo per assegnare un nome a una porzione di testo. Dopo che una macro è stata definita, il programmatore può scriverne il nome al posto della porzione di programma corrispondente. Una macro in realtà non è altro che un'abbreviazione per una porzione di testo. La Figura 7.4(a) mostra un programma in linguaggio assemblativo per il Pentium 4 che scambia due volte il contenuto delle variabili *p* e *q*. Queste sequenze possono essere definite come macro, come mostra la Figura 7.4(b). Dopo aver definito la macro, ogni occorrenza di *SWAP* viene sostituita dalle quattro linee:

```
MOV EAX,P
MOV EBX,Q
MOV Q,EAX
MOV P,EBX
```

Il programmatore ha definito *SWAP* come un'abbreviazione delle quattro istruzioni.

Anche se assemblatori diversi utilizzano notazioni leggermente differenti, in tutti i casi una definizione di macro è composta dalle seguenti parti principali:

1. un'intestazione che indica il nome della macro che si sta definendo
2. il testo che costituisce il corpo della macro
3. una pseudoistruzione che segna la fine della definizione (per esempio, ENDM).

Quando l'assemblatore incontra una definizione di macro, la salva nella tabella delle definizioni di macro per poterla utilizzare successivamente. Da quel momento in poi ogni volta che il nome della macro appare come codice operativo, l'assemblatore la sostituisce con il corpo della macro. L'uso del nome di una macro come codice operativo viene detto **chiamata di macro**, mentre la sua sostituzione con il corpo della macro è chiamata **espansione di macro**.

```
MOV   EAX,P
MOV   EBX,Q
MOV   Q,EAX
MOV   P,EBX

MOV   EAX,P
MOV   EBX,Q
MOV   Q,EAX
MOV   P,EBX
```

(a)

```
SWAP  MACRO
      MOV EAX,P
      MOV EBX,Q
      MOV Q,EAX
      MOV P,EBX
      ENDM

      SWAP

      SWAP
```

(b)

Figura 7.4 Codice in linguaggio assemblativo per scambiare due volte P e Q. Senza (a) e con (b) macro.

L'espansione della macro viene effettuata durante il processo assemblativo e non durante l'esecuzione del programma. Questo è un punto importante. Il programma della Figura 7.4(a) e quello della Figura 7.4(b) produrranno esattamente lo stesso codice in linguaggio macchina. Guardando solamente il programma tradotto in linguaggio macchina, è impossibile stabilire se, durante la generazione, siano state utilizzate o meno delle macro. Il motivo è che, una volta completata l'espansione delle macro, l'assemblatore elimina le loro definizioni, di cui non rimane alcuna traccia nel programma generato.

Le chiamate di macro non devono essere confuse con le chiamate di procedure. La differenza principale è che una chiamata di macro è un'istruzione diretta all'assemblatore, affinché sostituisca il nome della macro con il suo corpo. Una chiamata di procedura è invece un'istruzione macchina inserita nel programma oggetto e che sarà eseguita in un secondo momento per chiamare la procedura. La Figura 7.5 confronta le chiamate di macro con le chiamate di procedura.

Domanda	Macro	Procedura
Quando viene effettuata la chiamata?	Durante l'assemblaggio	Durante l'esecuzione del programma
Il corpo è inserito nel programma oggetto in ogni punto in cui avviene una chiamata?	Sì	No
L'istruzione per la chiamata di procedura è inserita nel programma oggetto ed eseguita successivamente?	No	Sì
Occorre usare un'istruzione di ritorno dopo aver effettuato la chiamata?	No	Sì
Quante copie del corpo appaiono nel programma oggetto?	Una per chiamata di macro	Una

Figura 7.5 Confronto tra le chiamate di macro e le chiamate di procedure.

Concettualmente conviene pensare che il processo assemblativo avvenga in due fasi. Nella prima vengono salvate tutte le definizioni delle macro e vengono espanse le loro chiamate. Nella seconda il testo risultante viene elaborato come se fosse il programma originale. Vedendolo in questo modo, il programma sorgente viene letto e poi trasformato in un altro programma nel quale sono state rimosse tutte le definizioni delle macro e le chiamate sono state sostituite dai loro corpi. L'output risultante è un programma in linguaggio assemblativo che non contiene alcuna macro e che può essere passato all'assemblatore.

È importante tenere a mente che un programma è semplicemente una stringa di caratteri che comprende lettere, cifre, spazi, segni di punteggiatura e "ritorni a capo" (passaggio a una nuova linea). L'espansione di una macro consiste nel sostituire una parte di questa stringa con una nuova stringa di caratteri. L'uso delle macro è una tecnica per manipolare stringhe di caratteri, senza preoccuparsi del loro significato.

7.2.2 Macro con parametri

Le macro descritte precedentemente sono utilizzabili in programmi di breve lunghezza in cui si ripete più volte una sequenza d'istruzioni perfettamente identica. Tuttavia è frequente il caso in cui un programma contiene varie sequenze d'istruzioni che non sono del tutto identiche. Nell'esempio della Figura 7.6(a) la prima sequenza scambia *P* con *Q*, mentre la seconda *R* con *S*.

Gli assemblatori di macro gestiscono queste situazioni consentendo che le definizioni delle macro specifichino dei **parametri formali** e che le chiamate forniscano dei **parametri attuali**. Quando una macro viene espansa i parametri formali che appaiono nel suo corpo vengono sostituiti dai corrispondenti parametri attuali. I parametri attuali sono specificati nel campo operandi della chiamata di macro. La Figura 7.6(b) mostra il programma della Figura 7.6(a) riscritto utilizzando una macro con due parametri. *P1* e *P2* sono i parametri formali. Nel momento in cui la macro viene espansa, ogni occorrenza di *P1* nel corpo della macro viene sostituita dal secondo parametro attuale. Nella chiamata di macro

```
CHANGE P, Q
```

```
MOV   EAX,P
MOV   EBX,Q
MOV   Q,EAX
MOV   P,EBX

MOV   EAX,R
MOV   EBX,S
MOV   S,EAX
MOV   R,EBX
```

(a)

```
CHANGE  MACRO P1, P2
        MOV EAX,P1
        MOV EBX,P2
        MOV P2,EAX
        MOV P1,EBX
        ENDM

        CHANGE P, Q

        CHANGE R, S
```

(b)

Figura 7.6 Sequenze d'istruzioni quasi identiche. Senza (a) e con (b) macro.

P è il primo parametro attuale e *Q* è il secondo. I programmi prodotti dalle due parti della Figura 7.6 sono quindi identici: contengono esattamente le stesse istruzioni con gli stessi operandi.

7.2.3 Caratteristiche avanzate

La maggior parte dei processori di macro mette a disposizione un gran numero di caratteristiche avanzate per rendere più facile la vita dei programmatori in linguaggio assemblativo. In questo paragrafo daremo un'occhiata ad alcune caratteristiche avanzate di MASM. Un problema che sorge usando gli assemblatori che supportano le macro è la duplicazione delle etichette. Supponiamo che una macro contenga un'istruzione di diramazione condizionale e un'etichetta che rappresenta la destinazione del salto. Se la macro viene chiamata due o più volte, l'etichetta verrà duplicata causando un errore del linguaggio assemblativo. Una soluzione consiste nell'obbligare il programmatore a fornire tramite parametro una diversa etichetta ogni volta che chiama la macro. La soluzione utilizzata da MASM è invece quella di permettere la dichiarazione di etichetta `LOCAL`, e lasciare che l'assemblatore generi un'etichetta diversa per ogni espansione della macro. Altri assemblatori utilizzano la regola secondo la quale le etichette numeriche sono automaticamente definite come locali.

MASM e la maggior parte degli assemblatori permettono di definire macro all'interno di altre macro. Questa funzionalità risulta particolarmente utile se usata insieme all'assemblaggio condizionale. In genere la stessa macro viene definita in entrambi i rami di un costrutto `IF`, come il seguente:

```
M1 MACRO
IF WORDSIZE GT 16
M2     MACRO
       ...
       ENDM
   ELSE
M2     MACRO
   ...
       ENDM
   ENDIF
       ENDM
```

In entrambi i casi la macro *M2* viene definita, ma la sua definizione dipende dal fatto che il programma venga assemblato su una macchina a 16 bit oppure su una a 32 bit. Se *M1* non viene chiamata, *M2* non viene definita.

Infine, le macro possono chiamare altre marco, comprese loro stesse. Se una macro è ricorsiva, cioè se richiama se stessa, deve passare un parametro modificato a ogni espansione e che fa terminare la ricorsione quando raggiunge un determinato valore. Se così non fosse, l'assemblatore potrebbe entrare in un ciclo infinito, e in tal caso l'utente dovrebbe interrompere esplicitamente l'assemblatore.

7.2.4 Implementazione delle funzionalità macro negli assemblatori

Per implementare una funzionalità macro un assemblatore deve essere in grado di eseguire due funzioni: salvare le definizioni delle macro ed espandere le loro chiamate. Esaminiamo una alla volta queste due operazioni.

L'assemblatore deve mantenere una tabella con tutti i nomi delle macro e, associato a ogni nome, un puntatore alla definizione precedentemente memorizzata, in modo che sia possibile recuperarla quando è necessario. Alcuni assemblatori hanno una tabella separata per i nomi delle macro, mentre altri sono dotati di un'unica tabella in cui sono mantenute, oltre alle macro, anche tutte le istruzioni macchina e le pseudoistruzioni.

Quando s'incontra una definizione di macro si aggiunge nella tabella un elemento che indica il nome della macro, il numero di parametri formali e un puntatore a un'altra tabella, la tabella delle definizioni delle macro, che conterrà il suo corpo. In questa fase si costruisce anche una lista dei parametri formali da utilizzare nell'elaborazione della definizione. Viene quindi letto il corpo della macro e memorizzato all'interno della tabella delle definizioni. I parametri formali che s'incontrano all'interno del corpo sono indicati mediante simboli speciali. Come esempio mostriamo la rappresentazione interna della definizione della macro *CHANGE*, utilizzando il punto e virgola come "ritorno a capo" e la "e commerciale" (&) come simbolo dei parametri formali:

```
MOV EAX,&P1; MOV EBX,&P2; MOV &P2,EAX; MOV &P1,EBX;
```

Nella tabella delle definizioni delle macro il corpo di ciascuna macro è semplicemente una stringa di caratteri.

Durante il primo passo del processo di assemblaggio, vengono cercati i codici operativi e le macro vengono espanse. Ogni volta che s'incontra una definizione di macro la si memorizza nella tabella delle macro. Quando viene chiamata una macro l'assemblatore interrompe temporaneamente la lettura dal dispositivo di input e inizia a leggere il corpo della macro precedentemente memorizzato. I parametri formali presenti al suo interno vengono poi sostituiti dai parametri attuali specificati dalla chiamata. La presenza del simbolo & prima dei parametri formali permette all'assemblatore di riconoscerli più facilmente.

7.3 Processo di assemblaggio

Nei paragrafi successivi descriveremo brevemente come funziona un assemblatore. Anche se ogni macchina ha il proprio linguaggio assemblativo, i diversi processi di assemblaggio sono tra loro sufficientemente simili da poterci permettere una descrizione generale.

7.3.1 Assemblatori a due passate

Dato che ogni programma in linguaggio assemblativo consiste in una serie d'istruzioni scritte su un'unica riga, di primo acchito potrebbe sembrare naturale avere un assemblatore che legge un'istruzione, la traduce in linguaggio macchina e scrive su file il codice generato, oltre a produrre, se richiesto, un altro file contenente il listato. Questo procedimento andrebbe ripetuto fino alla traduzione dell'intero programma. Purtroppo però questa strategia non funziona.

Consideriamo il caso in cui la prima istruzione sia una diramazione verso *L*. L'assemblatore non può assemblare questa istruzione finché non venga a conoscenza dell'indirizzo dell'istruzione *L*. Dato che questa istruzione potrebbe trovarsi verso la fine del programma l'assemblatore è obbligato a leggere praticamente tutto il programma per trovare il suo indirizzo. Il fatto di utilizzare un simbolo, *L*, prima ancora che sia stato definito, costituisce il

problema dei riferimenti in avanti; in altre parole si effettua un riferimento a un simbolo la cui definizione ha luogo in un punto successivo del programma.

È possibile gestire i riferimenti in avanti in due modi. Primo, l'assemblatore può leggere il programma sorgente due volte. Ciascuna lettura del programma sorgente è chiamata **passata** e i traduttori che leggono due volte il programma di input sono detti **a due passate**. Durante la prima lettura questi assemblatori raccolgono in una tabella tutte le definizioni dei simboli, comprese le etichette delle istruzioni. Prima di iniziare la seconda passata si conoscono quindi i valori di tutti i simboli. In tal modo non rimane alcun riferimento in avanti ed è possibile leggere ogni istruzione, assemblarla e generare il codice corrispondente. Questo approccio richiede una passata aggiuntiva per elaborare il programma di input, ma è concettualmente semplice.

Un secondo approccio consiste nel leggere il file in ingresso una sola volta, convertirlo in un formato intermedio e memorizzare in una tabella della memoria questa forma intermedia. In seguito viene effettuata una seconda passata sulla tabella, invece che sul codice sorgente come avveniva nel primo approccio. Se è disponibile una sufficiente quantità di memoria (fisica o virtuale) questo approccio risparmia il tempo dovuto alle operazioni di I/O. Se non viene richiesta la generazione del listato, è possibile ridurre al minimo indispensabile questa forma intermedia.

In entrambi gli approcci, la prima passata ha anche il compito di salvare tutte le definizioni di macro e di espanderle quando si incontrano le loro chiamate. Generalmente quindi la definizione dei simboli e l'espansione delle macro avvengono in un'unica passata.

7.3.2 Prima passata

La principale funzione della prima passata è quella di costruire la cosiddetta **tabella dei simboli**, contenente i valori di tutti i simboli. Un simbolo è un'etichetta, oppure un valore, al quale è stato assegnato un nome simbolico attraverso una pseudoistruzione, come

```
BUFSIZE EQU 8192
```

Per assegnare nel campo etichetta di un'istruzione un valore a un simbolo, l'assemblatore deve conoscere quale indirizzo avrà tale istruzione durante l'esecuzione del programma. Durante il processo di assemblaggio l'assemblatore mantiene una variabile, chiamata **ILC** (*Instruction Location Counter*, "contatore di locazioni delle istruzioni"), per tener traccia del-

Etichetta	Codice operativo	Operandi	Commenti	Lunghezza	ILC
MARIA:	MOV	EAX, I	EAX = I	5	100
	MOV	EBX, J	EBX = J	6	105
ROBERTA:	MOV	ECX, K	ECX = K	6	111
	IMUL	EAX, EAX	EAX = I × I	2	117
	IMUL	EBX, EBX	EBX = J × J	3	119
	IMUL	ECX, ECX	ECX = K × K	3	122
MARIANNA:	ADD	EAX, EBX	EAX = I × I + J × J	2	125
	ADD	EAX, ECX	EAX = I × I + J × J + K × K	2	127
STEFANIA:	JMP	DONE	salto a DONE	5	129

Figura 7.7 Il contatore di locazioni delle istruzioni (ILC) tiene traccia degli indirizzi delle locazioni di memoria in cui le istruzioni saranno caricate. In questo esempio le istruzioni prima di MARIA occupano 100 byte.

Identificatore	Valore	Altre informazioni
MARIA	100	
ROBERTA	111	
MARIANNA	125	
STEFANIA	129	

Figura 7.8 Tabella dei simboli per il programma della Figura 7.7.

l'indirizzo che l'istruzione che sta assemblando avrà a tempo di esecuzione. All'inizio della prima passata la variabile ha valore 0 e, come mostra la Figura 7.7, ogni volta che viene elaborata un'istruzione la variabile viene incrementata della sua lunghezza. L'esempio mostrato nella figura è relativo al Pentium 4 ed è più che sufficiente. Infatti nel caso di SPARC e di Motorola il procedimento è del tutto analogo. Inoltre, se venisse indetto un concorso per premiare il linguaggio assemblativo meno leggibile, quello di SPARC sarebbe il favorito.

Nella maggior parte degli assemblatori la prima passata utilizza tre tabelle interne per i simboli, le pseudoistruzioni e i codici operativi. Se necessario viene mantenuta anche la tabella dei letterali (si veda in seguito la definizione).

Come mostra la Figura 7.8, la tabella dei simboli ha una linea per ciascun identificatore (simbolo). I simboli vengono definiti quando sono utilizzati come etichette oppure mediante una definizione esplicita (per esempio, `EQU`). Ogni elemento della tabella dei simboli contiene il simbolo stesso (o un puntatore a esso), il suo valore numerico e, in alcuni casi, altre informazioni. Queste informazioni aggiuntive possono comprendere:

1. la lunghezza del campo dati associato al simbolo
2. i bit di rilocazione (il simbolo cambia valore se il programma è caricato in un indirizzo diverso rispetto a quello che ha assunto l'assemblatore?)
3. se il simbolo sia o meno accessibile dall'esterno della procedura.

La tabella dei codici operativi (la Figura 7.9 ne mostra una parte) contiene un elemento per ciascun codice simbolico (mnemonico) del linguaggio. Ciascun elemento contiene il codice simbolico, due operandi, il valore numerico del codice operativo, la lunghezza dell'istruzione

Codice operativo	Primo operando	Secondo operando	Codice esadecimale	Lunghezza dell'istruzione	Classe dell'istruzione
AAA	—	—	37	1	6
ADD	EAX	immed32	05	5	4
ADD	reg	reg	01	2	19
AND	EAX	immed32	25	5	4
AND	reg	reg	21	2	19

Figura 7.9 Parti della tabella dei codici operativi per un assemblatore del Pentium 4.

e un numero di tipo (che permette di raggruppare i codici operativi a seconda del numero e tipo di operandi).

Se per esempio un'istruzione ADD contiene come primo operando EAX e come secondo una costante a 32 bit (`immed32`), si utilizza il codice operativo 0x05 e la lunghezza dell'istruzione è di 5 byte. In realtà le costanti che possono essere espresse con 8 o 16 bit utilizzano codici operativi diversi, ma non sono mostrati nella figura. Se ADD viene utilizzata con due registri come operandi, allora la sua lunghezza risulta di 2 byte e il suo codice operativo è 0x01. La classe dell'istruzione 19 (il valore è arbitrario) dovrebbe essere assegnata a tutte le combinazioni codice operativo-operando che seguono le stesse regole. Le istruzioni facenti parte di questa classe dovrebbero quindi essere elaborate allo stesso modo dell'istruzione ADD che usa due registri come operandi. La classe dell'istruzione designa in realtà una procedura dell'assemblatore, richiamata per elaborare tutte le istruzioni di un certo tipo.

Alcuni assemblatori permettono ai programmatori di scrivere istruzioni che usano l'indirizzamento immediato anche se nel linguaggio destinazione non esiste alcuna istruzione corrispondente. Per gestire queste istruzioni, chiamiamole *pseudoimmediate*, l'assemblatore alloca alla fine del programma la memoria per l'operando immediato e genera un'istruzione che fa riferimento a esso. Per esempio il mainframe IBM 3090 non ha istruzioni immediate, ma i programmatori possono comunque scrivere

```
L 14,=F'5'
```

per caricare il registro 14 con una parola costante il cui valore è 5. In questo modo il programmatore non è obbligato a scrivere esplicitamente una pseudoistruzione per allocare una parola inizializzata a 5, indicando la sua etichetta e poi utilizzandola nell'istruzione L. Le costanti per le quali l'assemblatore riserva automaticamente la memoria sono chiamate **letterali**. I letterali, oltre a risparmiare un po' di scrittura di codice al programmatore, migliorano anche la leggibilità del programma rendendo visibile il valore delle costanti all'interno delle istruzioni. La prima passata dell'assemblatore dovrebbe costruire una tabella di tutti i letterali utilizzati nel programma. Le nostre tre macchine di esempio sono dotate d'istruzioni immediate e quindi i loro assemblatori non utilizzano i letterali. Oggigiorno è abbastanza comune che un assemblatore sia dotato d'istruzioni letterali, ma una volta non era così. È probabile che l'uso molto diffuso dei letterali abbia reso evidente agli occhi dei progettisti delle macchine che l'indirizzamento diretto sia una buona idea. Se però è necessario utilizzare i letterali, allora durante l'assemblaggio occorre mantenere una tabella in cui viene aggiunto un elemento ogni volta che si incontra un letterale. Dopo la prima passata questa tabella viene ordinata e gli elementi duplicati vengono rimossi.

La Figura 7.10 mostra una procedura che potrebbe servire come base per la prima passata di un assemblatore. Lo stile di programmazione è autoesplicativo: i nomi delle procedure sono stati scelti per fornire una buona indicazione della loro funzione. L'aspetto più importante è che la Figura 7.10 rappresenta abbastanza bene la struttura della prima passata. La procedura è sintetica, può essere compresa facilmente, e rende evidente quale deve essere il passo successivo, ovvero la scrittura del corpo delle procedure richiamate al suo interno.

Alcune di queste procedure saranno relativamente corte, come *check_for_symbol* che restituisce una stringa di caratteri rappresentante un simbolo, se è presente, oppure `null` in caso contrario. Altre procedure, come *get_length_of_type1* e *get_length_of_type2* potrebbero invece essere più lunghe e richiamare altre procedure. In generale il numero di tipi non sarà


```
public static void pass_one( ) {
   // Questa procedura è uno schema della prima passata di un semplice assemblatore.
   boolean more_input = true;                    // indicatore che interrompe la prima passata
   String line, symbol, literal, opcode;         // campi dell'istruzione
   int location_counter, length, value, type;    // variabili generiche
   final int END_STATEMENT = -2;                 // la fine dell'input

   location_counter = 0;                         // assembla la prima istruzione in 0
   initialize_tables( );                         // inizializzazione generale

   while(more_input) {                           // more_input viene impostato a falso da END
      line = read_next_line( );                  // preleva una linea di input
      length = 0;                                // # byte nell'istruzione
      type = 0;                                  // di quale tipo (formato) è l'istruzione

      if (line_is_not_comment(line)) {
         symbol = check_for_symbol(line);        // alla linea è associata un'etichetta?
         if (symbol != null)                     // se sì, si registra il simbolo e il valore
            enter_new_symbol(symbol, location_counter);
         literal = check_for_literal(line);      // la linea contiene un letterale?
         if (literal != null)                    // se sì, lo si inserisce nella tabella
            enter_new_literal(literal);

         // Si determina ora il tipo del codice operativo. _1 indica un codice operativo inesistente
         opcode = extract_opcode(line);          // localizza il nome mnemonico del codice operativo
         type = search_opcode_table(opcode);     // trova il formato, p.es. OP REG1,REG2
         if (type < 0)                           // se non è un codice operativo, è una pseudoistruzione?
            type = search_pseudo_table(opcode);
         switch(type) {                          // determina la lunghezza di questa istruzione
            case 1: length = get_length_of_type1(line); break;
            case 2: length = get_length_of_type2(line); break;
            // altri casi
         }
      }
      write_temp_file(type, opcode, length, line);   // informazioni utili per la seconda passata
      location_counter = location_counter + length;  // aggiorna loc_ctr
      if (type == END_STATEMENT) {                   // l'input è terminato?
         more_input = false;                         // se sì, si eseguono delle operazioni finali
         rewind_temp_for_pass_two( );                // come riportare al punto di inizio il file temporaneo
         sort_literal_table( );                      // e ordinare la tabella dei letterali
         remove_redundant_literals( );               // e rimuovere da questa i duplicati
      }
   }
}
```

Figura 7.10 Prima passata di un semplice assemblatore.

ovviamente limitato a due, ma dipenderà dal linguaggio che si sta assemblando e da quanti tipi d'istruzioni è composto.

Oltre a facilitarne la stesura, strutturare i programmi in questo modo presenta altri vantaggi. Se più persone lavorano alla scrittura dell'assemblatore le varie procedure possono essere distribuite fra tutti i programmatori. Inoltre tutti i dettagli (più difficoltosi) riguardanti la lettura dell'input rimangono nascosti nella procedura *read_next_line*. Se dovessero cambiare, per esempio a causa di una modifica del sistema operativo, sarebbe necessario modificare solamente una delle procedure ausiliarie, mentre non occorrerebbe effettuare alcuna modifica alla procedura *pass_one*.

Durante la prima passata viene analizzata ogni linea per trovare il codice operativo, determinare il suo tipo e calcolare la lunghezza dell'istruzione. Queste informazioni sono necessarie anche per la seconda passata e quindi può essere utile trascriverle esplicitamente in

modo da evitare, nella passata successiva, di analizzare nuovamente la linea da zero. Tuttavia, riscrivere il file di input genera un maggior numero di operazioni di I/O. La scelta tra effettuare più operazioni di I/O per eliminare una seconda analisi delle istruzioni oppure effettuare meno operazioni di I/O, ma più analisi delle linee di codice, è condizionata da vari fattori, tra cui la velocità della CPU rispetto a quella del disco e l'efficienza del file system. In questo esempio scriveremo un file temporaneo contenente il tipo, il codice operativo, la lunghezza e l'effettiva linea di input. La seconda passata legge questa linea al posto di quella presente nel file di input.

La prima passata termina quando viene letta la pseudoistruzione `END`. Se necessario, a questo punto possono essere ordinate le tabelle dei simboli e dei letterali. Quest'ultimi, dopo essere stati ordinati, possono essere controllati al fine di individuare, ed eventualmente rimuovere, gli elementi doppi.

7.3.3 Seconda passata

La funzione della seconda passata è la generazione del programma oggetto e l'eventuale stampa del listato. Oltre a ciò la seconda passata deve generare delle informazioni richieste dal linker per collegare in un unico file eseguibile le procedure assemblate in momenti distinti. La Figura 7.11 mostra una bozza della procedura che implementa la seconda passata.

Le operazioni compiute dalla seconda passata sono più o meno simili a quelle della prima: le linee vengono lette ed elaborate una alla volta. Dato che all'inizio di ogni linea abbiamo scritto (sul file temporaneo) il tipo, il codice operativo e la lunghezza, queste informazioni vengono lette in modo da risparmiare parte della fase di analisi dell'input. Il lavoro principale della generazione del codice è realizzato dalle procedure *eval_type1*, *eval_type2*, e così via. Ciascuna di queste funzioni gestisce un particolare schema, per esempio un codice operativo e due registri come operandi, genera il codice binario corrispondente all'istruzione e infine lo restituisce all'interno della variabile *code*. Questo codice viene poi scritto su file. Con ogni probabilità la funzione *write_output* accumula in memoria il codice binario generato e lo scrive su file in grosse porzioni per ridurre il traffico dati verso il disco.

L'istruzione sorgente originale e il codice oggetto generato da questa (in formato esadecimale) possono quindi essere stampati oppure memorizzati in un buffer per rimandarne la stampa. Dopo aver modificato ILC la seconda passata preleva l'istruzione successiva.

Finora si è assunto che il programma sorgente non contenga alcun errore. Chiunque abbia mai scritto un programma, in qualsiasi linguaggio, sa bene quanto sia realistica una simile affermazione. Alcuni fra gli errori più comuni sono:

1. un simbolo è stato utilizzato senza essere stato definito
2. un simbolo è stato definito più di una volta
3. il nome nel campo del codice operativo non è un codice operativo lecito
4. un codice operativo non è seguito da un numero sufficiente di operandi
5. un codice operativo è seguito da un numero eccessivo di operandi
6. un numero ottale contiene le cifre 8 o 9


```
public static void pass_two( ) {
   // Questa procedura è uno schema della seconda passata di un semplice assemblatore.
   booleanmore_input = true;                    // indicatore che interrompe la seconda passata
   String line, opcode;                         // campi dell'istruzione
   int location_counter, length, type;          // variabili varie
   final int END_STATEMENT = -2;                // segnala la fine dell'input
   final int MAX_CODE = 16;                     // numero massimo di byte di codice per istruzione
   byte code [ ] = new byte[MAX_CODE ];         // contiene il codice generato da un'istruzione

   location_counter = 0;                        // assembla la prima istruzione in 0

   while (more_input) {                         // more_input viene impostato a falso da END
      type = read_type( );                      // preleva il campo tipo della linea successiva
      opcode = read_opcode( );                  // preleva il campo codice operativo della linea successiva
      length = read_length( );                  // preleva il campo della lunghezza della linea successiva
      line = read_line( );                      // preleva la linea attuale dell'input

      if (type! = 0) {                          // il tipo 0 è per le linee di commento
         switch(type) {                         // genera il codice di output
            case 1: eval_type1(opcode, length, line, code); break;
            case 2: eval_type2(opcode, length, line, code); break;
            // altri casi
         }
      }
      write_output(code);                       // scrive il codice binario
      write_listing(code, line);                // stampa una linea del listato
      location_counter = location_counter + length;  // aggiorna loc_ctr
      if (type == END_STATEMENT) {              // l'input è terminato?
         more_input = false;                    // se sì, si eseguono alcune operazioni finali
         finish_up( );                          // varie e conclusione
      }
   }
}
```

Figura 7.11 Seconda passata di un semplice assemblatore.

7. un registro è stato utilizzato in modo scorretto (per esempio, una diramazione verso un registro)
8. manca la pseudoistruzione `END`.

I programmatori sono molto ingegnosi nel trovare sempre nuovi tipi di errori da commettere. Quelli dovuti ai simboli non definiti spesso sono originati da errori di battitura e quindi un assemblatore astuto potrebbe provare a sostituire il simbolo non definito con quello che, fra quelli che sono stati correttamente definiti, più gli assomiglia. Per la maggior parte degli altri errori però c'è ben poco da fare. Quando un assemblatore incontra un'istruzione errata si limita a stampare un messaggio di errore e cerca di continuare l'assemblaggio.

7.3.4 Tabella dei simboli

Durante la prima passata l'assemblatore accumula informazioni sui simboli e i loro valori. Tali informazioni devono essere memorizzate nella tabella dei simboli per essere utilizzate durante la seconda passata. Ora prenderemo brevemente in esame alcuni tra i vari modi di organizzare la tabella dei simboli. In tutti i casi si cerca di simulare una **memoria associa-**

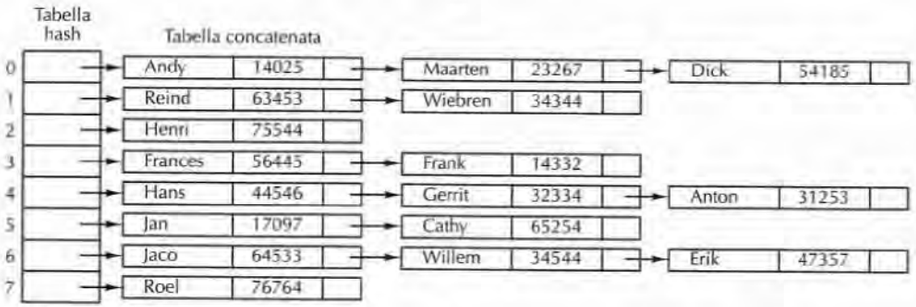
tiva, che concettualmente non è altro che un insieme di coppie (simbolo, valore). Dato un simbolo, la memoria associativa deve fornire il valore corrispondente.

La tecnica più semplice consiste nell'implementare la tabella dei simboli come un vettore di coppie, in cui il primo elemento è il simbolo (oppure un puntatore a esso) e il secondo è il valore (oppure un puntatore a esso). Quando si vuole recuperare un simbolo, la routine della tabella dei simboli effettua semplicemente una ricerca lineare all'interno dell'array finché non trova l'elemento desiderato. Questo metodo è facile da programmare, ma allo stesso tempo è lento, dato che per ciascuna ricerca occorre esaminare, in media, metà della tabella.

Un altro modo per organizzare la tabella dei simboli è quello di ordinarla rispetto ai simboli e di utilizzare un algoritmo di **ricerca dicotomica** per cercare il simbolo desiderato. Questo algoritmo funziona confrontando il simbolo con l'elemento centrale della tabella. Se il simbolo precede alfabeticamente l'elemento centrale della tabella, occorrerà continuare la ricerca nella prima metà della tabella. Se invece il simbolo segue l'elemento centrale, occorrerà cercarlo nella seconda metà della tabella. La procedura termina quando l'elemento centrale della tabella è uguale al simbolo cercato.

Andy	14025	0
Anton	31253	4
Cathy	65254	5
Dick	54185	0
Erik	47357	6
Frances	56445	3
Frank	14332	3
Gerrit	32334	4
Hans	44546	4
Henri	75544	2
Jan	17097	5
Jaco	64533	6
Maarten	23267	0
Reind	63453	1
Roel	76764	7
Willem	34544	6
Wiebren	34344	1

(a)



(b)

Figura 7.12 Codifica hash. (a) Simboli, valori e codici hash derivati dai simboli. (b) Tabella hash a otto elementi con liste concatenate di simboli e valori.

Assumendo che l'elemento centrale non sia uguale al simbolo cercato, possiamo comunque sapere in quale metà della tabella individuarlo. Si può quindi applicare nuovamente la ricerca dicotomica nella metà corretta della tabella; ciò permetterà di trovare il simbolo oppure di conoscere in quale quarto della tabella si trovi. Applicando ricorsivamente l'algoritmo è possibile completare una ricerca all'interno di una tabella di n elementi in $\log_2 n$ tentativi. Questa tecnica è ovviamente molto più veloce rispetto alla ricerca lineare, ma richiede che sia conservato l'ordine degli elementi della tabella.

Un modo completamente differente per simulare una memoria associativa è la tecnica conosciuta con il nome di **codifica hash** o **hashing**. In questo approccio occorre definire una funzione "hash" che mappa i simboli nell'intervallo di interi compreso tra 0 e $k-1$. Una possibile funzione consiste nel moltiplicare tra loro i codici ASCII dei caratteri del simbolo, ignorando un eventuale overflow, e considerando come risultato il resto della divisione per k di questo valore oppure il risultato della divisione rispetto a un numero primo. In realtà può andar bene qualsiasi funzione dell'input che fornisca una distribuzione uniforme dei valori hash. I simboli possono essere memorizzati in una tabella composta da k **secchielli** (*bucket*) numerati da 0 a $k-1$. Ogni posizione i della tabella hash punta a una lista concatenata in cui sono memorizzate le coppie (simbolo, valore) il cui valore hash del simbolo vale i. Con n simboli e una tabella hash con k posizioni, una lista avrà una lunghezza media pari a n/k. Se si sceglie k approssimativamente uguale a n, allora in media è possibile localizzare i simboli con un unico passo di ricerca. Modificando k è possibile ridurre la dimensione della tabella a spese però della velocità dell'operazione di ricerca. La codifica hash è illustrata nella Figura 7.12.

7.4 Collegamento e caricamento

La maggior parte dei programmi è composta da più di una procedura. Generalmente i compilatori e gli assemblatori traducono una procedura alla volta e memorizzano su disco il risultato della traduzione. Prima che il programma possa essere eseguito è necessario recuperare tutte le procedure e collegarle fra loro in modo appropriato. Inoltre, in assenza di memoria virtuale, occorre caricare in memoria centrale il programma ottenuto dal collegamento delle procedure. I programmi che eseguono questi passi sono chiamati in vari modi, tra cui ***linker***, ***linking loader*** e ***linkage editor***. La traduzione completa di un programma sorgente richiede due passi distinti, com'è mostrato nella Figura 7.13:

1. compilazione o assemblaggio delle procedure sorgenti
2. collegamento dei moduli oggetto.

Il primo passo viene eseguito dal compilatore o dall'assemblatore, e il secondo dal linker.

La traduzione che trasforma una procedura sorgente in un modulo oggetto costituisce un cambiamento di livello, dato che il linguaggio sorgente e quello destinazione hanno un diverso insieme d'istruzioni e usano una notazione differente. Il processo di collegamento invece non rappresenta un cambiamento di livello, dato che sia l'input del linker sia il suo output sono due programmi per la stessa macchina virtuale. La funzione del linker è quella di unire le procedure tradotte separatamente e di collegarle tra loro in modo da poterle eseguire come un'unica unità chiamata **programma eseguibile binario**. In MS-DOS, Windows

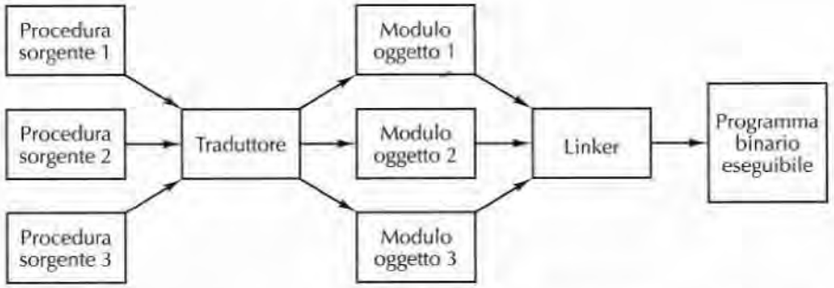


Figura 7.13 Generazione tramite un linker di un programma eseguibile binario a partire da un gruppo di procedure sorgente tradotte indipendentemente.

95/98 e NT i moduli oggetto sono caratterizzati dall'estensione *.obj* e i programmi eseguibili binari dall'estensione *.exe*. In UNIX i moduli oggetto hanno invece l'estensione *.o*, mentre i programmi eseguibili binari non hanno alcuna estensione.

Esiste un buon motivo per cui i compilatori e gli assemblatori traducono ciascuna procedura sorgente come un'entità separata. Se un compilatore, o un assemblatore, dovesse leggere una serie di procedure sorgente e produrre direttamente un programma in linguaggio macchina pronto per essere eseguito, allora la modifica anche solo di un'istruzione in una delle procedure sorgente richiederebbe una nuova traduzione di tutte le altre procedure.

Se si usa la tecnica dei moduli oggetto separati (mostrata nella Figura 7.13) è necessario ritradurre soltanto la procedura modificata e non quelle rimaste invariate, anche se occorre ricollegare nuovamente tra loro i moduli oggetto. Di solito la fase di collegamento è molto più veloce della traduzione e permette quindi di risparmiare una grande quantità di tempo durante lo sviluppo di un programma. Questo guadagno risulta particolarmente importante quando si hanno centinaia o migliaia di moduli.

7.4.1 Compiti del linker

Nel processo assemblativo il contatore di locazioni delle istruzioni viene impostato a 0 all'inizio della prima passata. Questo passo corrisponde ad assumere che il modulo oggetto sarà collocato, durante l'esecuzione, all'indirizzo (virtuale) 0. La Figura 7.14 mostra quattro moduli oggetto per una macchina generica. In questo esempio ciascun modulo inizia con un'istruzione `BRANCH` che porta, all'interno del modulo, all'istruzione `MOVE`.

Per poter eseguire il programma il linker porta in memoria centrale i moduli oggetto al fine di formare l'immagine del programma eseguibile binario, come mostra la Figura 7.15(a). L'idea è di creare all'interno del linker un'immagine esatta dello spazio d'indirizzamento virtuale del programma eseguibile e di posizionare tutti i moduli oggetto nelle locazioni corrette. Se non c'è sufficiente memoria per formare l'immagine è possibile utilizzare un file del disco. In genere una piccola sezione della memoria a partire dall'indirizzo zero viene utilizzata per i vettori di interrupt, la comunicazione con il sistema operativo, la cattura dei puntatori non inizializzati e altri scopi. Per questo motivo i programmi di solito iniziano a partire da una locazione maggiore di 0. Nella figura abbiamo (arbitrariamente) fatto iniziare i programmi all'indirizzo 100.

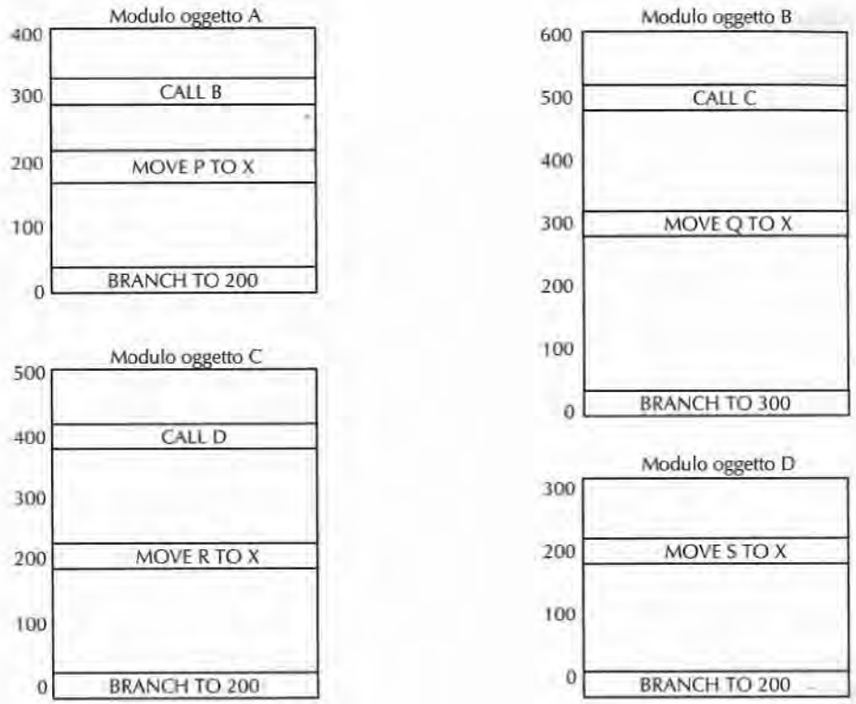


Figura 7.14 Ogni modulo ha il proprio spazio d'indirizzi, che inizia da 0.

Il programma della Figura 7.15(a) non è ancora pronto per essere eseguito, anche se è caricato all'interno dell'immagine del file eseguibile binario. Consideriamo che cosa succederebbe se l'esecuzione cominciasse con l'istruzione che si trova all'inizio del modulo *A*. Il programma non effettuerebbe correttamente il salto all'istruzione `MOVE`, dato che quella istruzione si trova ora all'indirizzo 300. In realtà tutte le istruzioni che fanno riferimento alla memoria fallirebbero per la stessa ragione. Occorre chiaramente trovare una soluzione al problema.

Il **problema della rilocazione** si verifica perché i moduli della Figura 7.14 hanno spazi degli indirizzi separati. Su una macchina con uno spazio degli indirizzi segmentato, come il Pentium 4, ogni modulo potrebbe teoricamente avere il proprio spazio degli indirizzi all'interno del proprio segmento. In ogni caso l'unico sistema operativo per Pentium 4 che supporta questa possibilità è OS/2; tutte le versioni di Windows e di Unix supportano solamente uno spazio degli indirizzi lineare e quindi tutti i moduli oggetto devono essere uniti fra loro al suo interno.

Inoltre neanche le istruzioni per la chiamata di procedure della Figura 7.15(a) funzionano correttamente. Nella figura si vede che all'indirizzo 400 il programmatore prova a chiamare il modulo oggetto *B*; l'assemblatore però non ha modo di sapere quale indirizzo inserire nell'istruzione `CALL B`, dato che ciascuna procedura è stata tradotta indipendentemente. L'indirizzo del modulo oggetto *B* non è infatti conosciuto fino al momento del col-

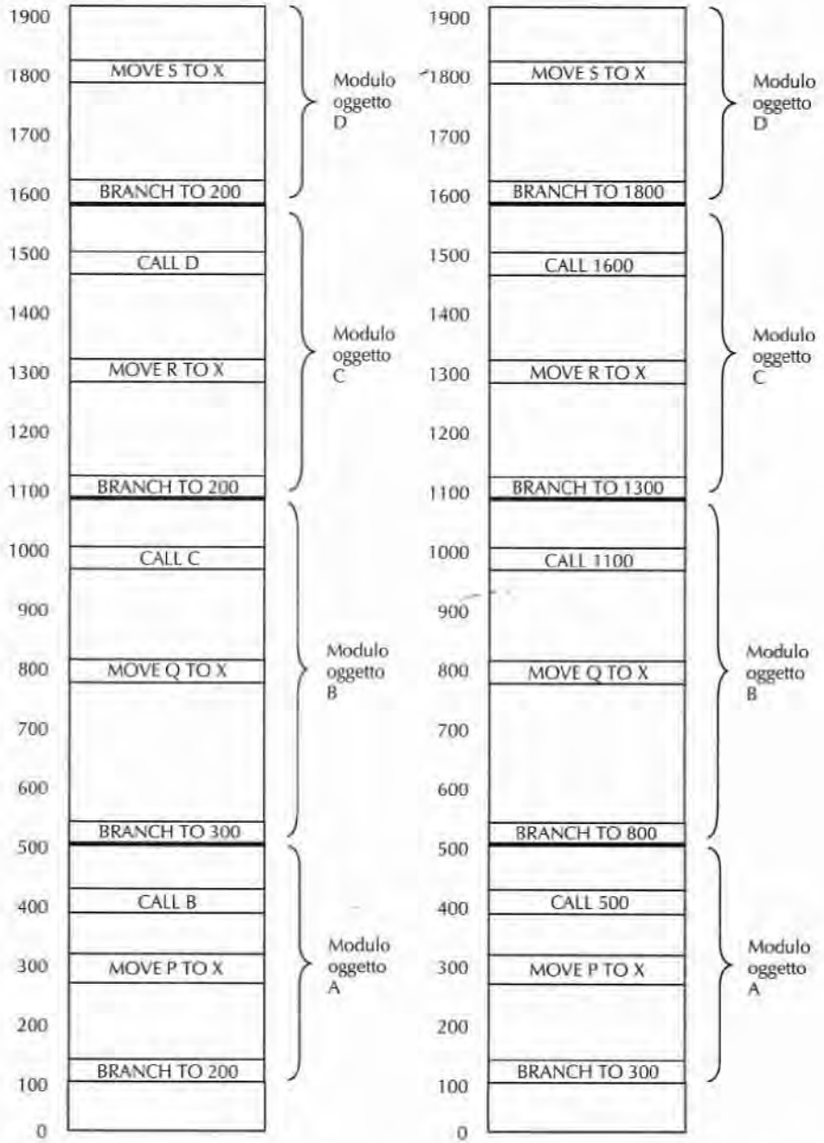


Figura 7.15 (a) I moduli oggetto della Figura 7.14 dopo essere stati posizionati nell'immagine binaria, ma prima di essere rilocati e collegati. (b) Gli stessi moduli oggetto dopo aver effettuato il collegamento e la rilocazione.

legamento. Questo è il **problema dei riferimenti esterni**. I due problemi possono essere risolti in modo semplice dal linker.

Il linker unisce gli spazi degli indirizzi separati dei diversi moduli oggetto all'interno di un unico spazio lineare degli indirizzi seguendo questi quattro passi:

1. costruisce una tabella di tutti i moduli oggetto e delle loro lunghezze
2. in base a questa tabella assegna un indirizzo di partenza per ciascun modulo
3. cerca tutte le istruzioni che fanno riferimento alla memoria e aggiunge a ciascuna di loro una **costante di rilocazione** uguale all'indirizzo di partenza del suo modulo
4. cerca tutte le istruzioni che fanno riferimento ad altre procedure e inserisce in quei punti gli indirizzi delle procedure corrispondenti.

La tabella seguente fornisce nome, lunghezza e indirizzo iniziale di tutti i moduli della Figura 7.15.

Modulo	Lunghezza	Indirizzo iniziale
A	400	100
B	600	500
C	500	1100
D	300	1600

7.4.2 Struttura di un modulo oggetto

Spesso i moduli oggetto sono composti da sei parti, come mostra la Figura 7.16. La prima parte contiene il nome del modulo, informazioni necessarie al linker (come le lunghezze delle singole parti del modulo), e in alcuni casi la data di assemblaggio.

La seconda parte è una lista di simboli definiti all'interno del modulo al quale possono fare riferimento altri moduli; i simboli sono associati ai valori corrispondenti. Se per esempio il modulo consiste di una procedura chiamata *bigbug*, il primo elemento della tabella conterrà la stringa di caratteri "bigbug" seguita dall'indirizzo nel quale si trova la procedura. Il programmatore in linguaggio assemblativo utilizza la pseudoistruzione `PUBLIC` della Figura 7.3 per indicare quali simboli devono essere dichiarati come **punti d'ingresso**.

Figura 7.16 Struttura interna di un modulo oggetto prodotto da un traduttore.

La terza parte del modulo oggetto comprende una lista di simboli utilizzati nel modulo, ma definiti in un altro, e anche una lista che indica quali simboli sono utilizzati dalle varie istruzioni macchina. Questa lista permette al linker di inserire gli indirizzi corretti nelle istruzioni che usano simboli esterni. Una procedura può chiamare altre procedure, tradotte indipendentemente, dichiarando come esterni i loro nomi. Il programmatore in linguaggio assemblativo indica quali simboli devono essere dichiarati come **simboli esterni** utilizzando una pseudoistruzione (per esempio la `EXTERN` della Figura 7.3). In alcuni calcolatori i punti d'ingresso e i riferimenti esterni sono combinati in un'unica tabella.

La quarta parte del modulo oggetto è costituita dal codice assemblato e dalle costanti. Questa è l'unica parte del modulo oggetto che verrà caricata in memoria per essere eseguita. Le altre parti saranno invece utilizzate dal linker ed eliminate prima che inizi l'esecuzione.

La quinta parte è il dizionario di rilocazione. Com'è possibile vedere nella Figura 7.15 occorre aggiungere una costante di rilocazione alle istruzioni contenenti dei riferimenti a indirizzi di memoria. Il linker, ispezionando la quarta parte del modulo, non ha modo di sapere quali parole di dati contengano istruzioni macchina e quali contengano costanti; questa tabella gli fornisce le informazioni sugli indirizzi che devono essere rilocati. Queste informazioni potrebbero formare una tabella di bit, con un bit per ogni indirizzo potenzialmente da rilocare, oppure una lista esplicita degli indirizzi da rilocare.

La sesta parte è composta da un identificatore della fine del modulo, dall'indirizzo a partire dal quale bisogna iniziare l'esecuzione e, in alcuni casi, da una checksum per rilevare gli errori che possono essere avvenuti durante la lettura del modulo.

La maggior parte dei linker richiede due passate. Durante la prima il linker legge tutti i moduli oggetto e costruisce una tabella con i nomi e le lunghezze dei moduli, oltre a una tabella globale di simboli contenenti tutti i punti d'ingresso e i riferimenti esterni. Durante la seconda passata i moduli oggetto vengono letti, rilocati e collegati uno alla volta.

7.4.3 Rilocazione a tempo del binding e dinamica

In un sistema multiprogrammato un programma può essere letto e portato nella memoria centrale, eseguito per un breve periodo di tempo, riportato su disco e poi riletto per essere eseguito un'altra volta. In un sistema di grandi dimensioni, quando è attivo un numero elevato di programmi, è difficile avere la certezza che il programma venga letto e caricato ogni volta nelle stesse locazioni.

La Figura 7.17 mostra che cosa succederebbe se il programma già rilocato della Figura 7.15(b) venisse caricato all'indirizzo 400 invece che all'indirizzo 100 in cui il linker l'aveva inserito originariamente. Tutti gli indirizzi di memoria risulterebbero errati e, per di più, non si potrebbero riutilizzare le informazioni di rilocazione, cancellate molto tempo prima. Anche se queste fossero ancora disponibili, il costo per rilocare tutti gli indirizzi ogni volta che un programma viene riportato in memoria centrale sarebbe troppo alto.

Il problema di spostare programmi che sono già stati collegati e rilocati è profondamente connesso al momento in cui viene completato il collegamento finale tra i nomi simbolici e gli indirizzi assoluti della memoria fisica. Un programma appena scritto contiene dei nomi simbolici per gli indirizzi di memoria, per esempio `BR L`. Il momento nel quale si determina l'effettivo indirizzo della memoria centrale corrispondente a *L* è chiamato **tempo del binding** ("tempo del collegamento").

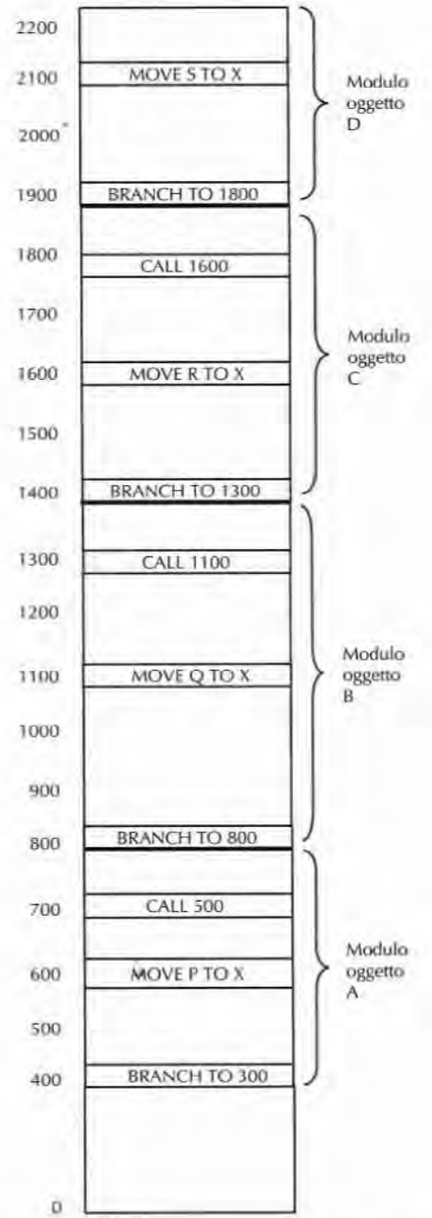


Figura 7.17 Programma binario rilocato della Figura 7.15(b) spostato all'indirizzo 300. Molte istruzioni si riferiscono ora a indirizzi di memoria errati.

Esistono almeno sei possibilità:

1. quando il programma viene scritto
2. quando il programma viene tradotto
3. quando il programma viene collegato, ma prima che sia caricato
4. quando il programma viene caricato
5. quando viene caricato un registro base utilizzato per l'indirizzamento
6. quando viene eseguita l'istruzione contenente l'indirizzo.

Se un'istruzione contenente un indirizzo di memoria viene spostata dopo il collegamento, il suo riferimento alla memoria risulta errato (assumendo che anche l'oggetto al quale fa riferimento sia stato spostato). Se il traduttore produce come output un eseguibile binario, il momento del collegamento coincide con quello della traduzione e in questo caso il programma deve essere eseguito all'indirizzo assunto dal traduttore. Il metodo descritto nel paragrafo precedente lega i nomi simbolici agli indirizzi assoluti durante la fase di collegamento; questa è la ragione per cui gli spostamenti dei programmi effettuati dopo il collegamento falliscono (come mostra l'esempio della Figura 7.17).

La questione solleva due tipi di problemi. Il primo riguarda il momento in cui i nomi simbolici vengono legati agli indirizzi virtuali, mentre il secondo riguarda il momento in cui gli indirizzi virtuali vengono legati agli indirizzi fisici. Il collegamento si può dire completato solo dopo che queste due operazioni sono state terminate. Quando il linker riunisce i diversi spazi degli indirizzi crea in realtà uno spazio virtuale di indirizzi. La rilocazione e il collegamento hanno l'effetto di collegare i nomi simbolici con specifici indirizzi virtuali. Questa osservazione è vera a prescindere dal fatto che venga o meno utilizzata la memoria virtuale.

Assumiamo per il momento che lo spazio degli indirizzi della Figura 7.15(b) sia paginato. È chiaro che gli indirizzi virtuali corrispondenti ai nomi simbolici *A*, *B*, *C* e *D* sono già stati determinati, anche se i loro indirizzi fisici dipenderanno, in ogni momento in cui verranno utilizzati, dal contenuto della tabella delle pagine. Un programma eseguibile binario è in realtà un collegamento tra nomi simbolici e indirizzi virtuali.

Qualsiasi meccanismo che permetta di modificare facilmente la corrispondenza degli indirizzi virtuali negli indirizzi fisici della memoria faciliterà lo spostamento dei programmi all'interno della memoria principale, anche dopo che essi sono stati legati a uno spazio degli indirizzi virtuali. Uno di questi meccanismi è la paginazione. Dopo che un programma è stato spostato all'interno della memoria centrale occorre modificare solamente la sua tabella delle pagine e non il programma stesso.

Una seconda tecnica consiste nell'uso di un registro di rilocazione durante l'esecuzione (come avveniva per il CDC 6600 e i suoi successori). Sulle macchine che utilizzano questa tecnica il registro punta sempre all'indirizzo fisico della memoria in cui inizia il programma corrente. Il registro di rilocazione viene sommato via hardware a tutti gli indirizzi, prima che essi siano inviati alla memoria. L'intero processo di rilocazione risulta così trasparente ai programmi utente. Quando viene spostato un programma il sistema operativo deve aggiornare il registro di rilocazione. Questo meccanismo è però meno generale rispetto alla paginazione, dato che l'intero programma deve essere spostato come una singola unità (oppure come due, nel caso in cui vi siano registri di rilocazione separati per il codice e per i dati, come nel processore Intel 8088).

Sulle macchine in grado di effettuare riferimenti alla memoria relativi al contatore d'istruzioni è possibile ricorrere a un terzo meccanismo. Grazie a questa funzionalità è possibile sfruttare il fatto che molte istruzioni di salto sono relative al valore di PC. Ogni volta che si sposta un programma nella memoria centrale occorre modificare solamente il PC. Un programma in cui tutti i riferimenti alla memoria sono relativi al PC oppure sono assoluti (per esempio, gli indirizzi assoluti dei registri dei dispositivi di I/O) è detto **indipendente dalla posizione**. Una procedura indipendente dalla posizione può essere collocata in qualsiasi punto dello spazio degli indirizzi virtuali senza bisogno di effettuare la rilocazione.

7.4.4 Collegamento dinamico

Nella strategia di collegamento descritta nel Paragrafo 7.4.1 tutte le procedure che un programma potrebbe richiamare sono collegate prima che abbia inizio l'esecuzione. Se si completano tutti i collegamenti prima dell'esecuzione, non si trae pieno vantaggio dalla memoria virtuale, di cui un calcolatore può essere dotato. Molti programmi contengono procedure, chiamate soltanto in particolari, e infrequenti, circostanze. I compilatori, per esempio, hanno alcune procedure per tradurre istruzioni utilizzate raramente, e altre il cui compito è quello di gestire condizioni di errore che si verificano di rado.

Un modo più flessibile per collegare procedure compilate separatamente è quello di collegarle nel momento in cui ciascuna procedura viene caricata. Questo processo è conosciuto con il nome di **collegamento dinamico**. Il primo calcolatore che ha impiegato questa tecnica è stato MULTICS, la cui pionieristica implementazione è, sotto alcuni aspetti, ancora ineguagliata. Nei paragrafi successivi analizzeremo come avviene il collegamento dinamico in alcuni sistemi.

Collegamento dinamico in MULTICS

L'implementazione di MULTICS del collegamento dinamico associa a ciascun programma un **segmento di collegamento** contenente un blocco d'informazioni per ciascuna procedura del programma. Questo blocco d'informazioni inizia con una parola riservata per l'indirizzo virtuale della procedura, seguita dal nome della procedura, memorizzato come stringa di caratteri.

Quando si utilizza il collegamento dinamico le chiamate di procedura nel linguaggio sorgente vengono tradotte in istruzioni che indirizzano, in modo indiretto, la prima parola del blocco di collegamento (la Figura 7.18(a) ne mostra un esempio). Il compilatore riempie questa parola o con un indirizzo non valido oppure con una speciale stringa di bit che determina un'eccezione.

Quando si richiama una procedura che fa parte di un altro segmento, il tentativo di indirizzare la parola non valida solleva nel linker dinamico un'eccezione. Il linker legge quindi la stringa di caratteri che si trova nella parola che segue l'indirizzo non valido e cerca una procedura con questo nome all'interno della directory dell'utente. Viene quindi assegnato un indirizzo virtuale a questa procedura, di solito all'interno del proprio segmento privato; inoltre, come mostra la Figura 7.18(b), questo indirizzo virtuale viene scritto nel segmento di collegamento, al posto dell'indirizzo non valido. Successivamente viene rieseguita l'istruzione che ha provocato l'errore di collegamento e ciò permette al programma di continuare la propria esecuzione dal punto in cui si trovava prima che venisse sollevata l'eccezione.

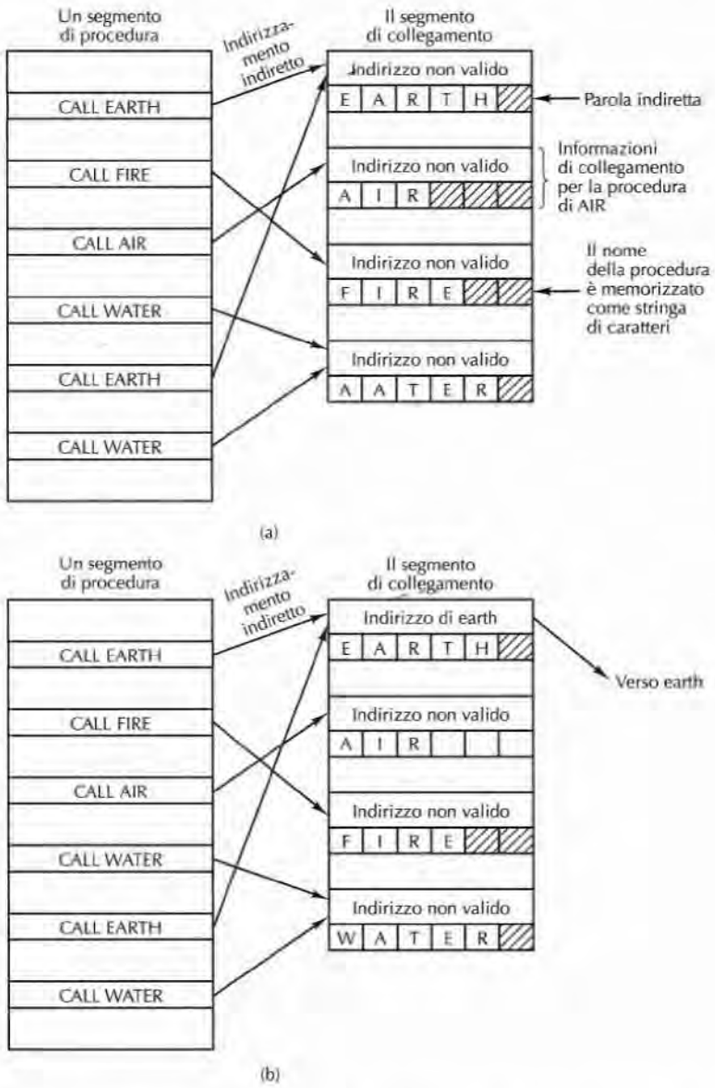


Figura 7.18 Collegamento dinamico. (a) Prima che *EARTH* sia chiamata. (b) Dopo che *EARTH* è stata chiamata e collegata.

Tutti i successivi riferimenti alla procedura avverranno senza provocare errori di collegamento, dato che la prima parola del segmento di collegamento contiene ora un indirizzo virtuale valido. Da ciò ne consegue che il linker dinamico viene invocato solamente la prima volta in cui viene richiamata una procedura e non durante le chiamate successive.

Collegamento dinamico in Windows

Tutte le versioni del sistema operativo Windows, compreso NT, supportano e sfruttano pienamente il collegamento dinamico. Allo scopo viene utilizzato un formato di file chiamato **DLL** (*Dynamic Link Library*, "libreria a collegamento dinamico"). Le DLL possono contenere procedure, dati oppure entrambi, e sono comunemente utilizzate per permettere a due o più processi di condividere librerie di dati e procedure. Molte DLL hanno l'estensione *.dll*, ma vengono utilizzate anche estensioni diverse, come *.drv* (per le librerie di driver) e *.fon* (per le librerie di caratteri).

Nella sua forma più comune una DLL è una libreria composta da un gruppo di procedure che possono essere caricate in memoria e alle quali possono accedere più processi nello stesso istante. La Figura 7.19 rappresenta due programmi che condividono un file DLL contenente quattro procedure, *A*, *B*, *C* e *D*. Nell'illustrazione il programma 1 usa la procedura *A* e il programma 2 usa la *C*, anche se i due programmi avrebbero comunque potuto utilizzare la stessa procedura.

Il linker costruisce una DLL a partire da un insieme di file di input. La costruzione di una DLL è in pratica molto simile alla creazione di un programma eseguibile binario, tranne che per uno speciale indicatore che viene specificato al linker in modo che crei una DLL. Di solito le DLL vengono costruite a partire da gruppi di librerie di procedure che ci si aspetta verranno utilizzate da più processi. Le procedure d'interfaccia alla libreria delle chiamate di sistema di Windows e le librerie grafiche di grandi dimensioni sono due esempi di DLL molto comuni. L'uso delle DLL ha il vantaggio di risparmiare spazio in memoria e sul disco. Se una libreria usata molto frequentemente venisse collegata staticamente a ciascun programma che ne fa uso, essa comparirebbe all'interno di molti programmi binari eseguibili sia in memoria sia sul disco, sprecando una gran quantità di spazio. Facendo ricorso alle DLL ciascuna libreria è presente una sola volta in memoria e una sola sul disco.

Oltre a risparmiare spazio questo approccio agevola l'aggiornamento delle procedure, anche dopo che i programmi che le utilizzano sono stati compilati e collegati. Per pacchetti

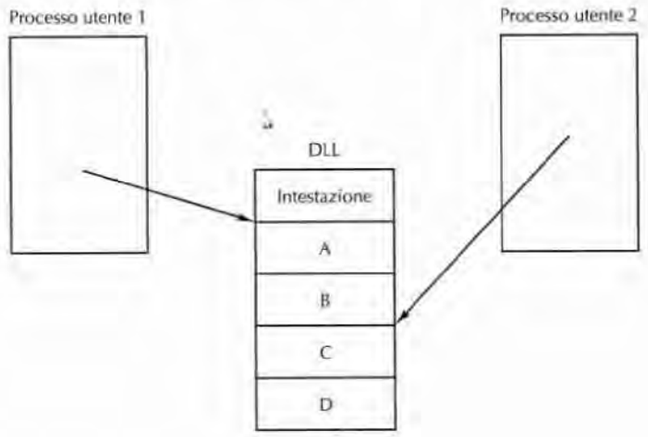


Figura 7.19 Un file DLL usato da due processi.

software commerciali, di cui raramente gli utenti hanno a disposizione il codice, l'uso delle DLL permette al venditore di correggere i bug presenti nelle librerie semplicemente ridistribuendo via Internet nuovi file DLL, senza dover cambiare in alcun modo il programma binario principale.

La differenza principale tra una DLL e un eseguibile binario è che la DLL non può essere eseguita da sola (non ha al suo interno un programma principale). Inoltre essa contiene nella sua intestazione informazioni di tipo diverso e possiede delle procedure aggiuntive che non sono legate alle procedure della libreria. È presente per esempio una procedura che viene automaticamente chiamata ogni volta che un nuovo processo è collegato alla DLL e un'altra che viene automaticamente chiamata quando un processo viene invece scollegato. Queste procedure possono allocare o deallocare aree di memoria o gestire altri tipi di risorse richieste dalla DLL.

Ci sono due modi per collegare un programma a una DLL. Nel primo, chiamato **collegamento implicito**, il programma dell'utente viene collegato a un file speciale chiamato **libreria importata,** generato da un programma di utilità che estrae alcune informazioni dalla DLL. La libreria importata costituisce il collante che permette al programma utente di accedere alla DLL. Un programma utente può essere collegato a più librerie importate. Quando viene caricato un programma che utilizza il collegamento implicito, Windows lo esamina per vedere quali DLL utilizza e controlla che tutte siano presenti in memoria. Quelle che non lo sono vengono caricate immediatamente (ma non necessariamente nella loro interezza, dato che sono paginate). Vengono apportati alcuni cambiamenti alle strutture dati delle librerie importate in modo da poter localizzare le procedure richiamate (queste modifiche sono analoghe a quelle viste nella Figura 7.18). Inoltre esse devono essere mappate nello spazio degli indirizzi virtuali del programma. A questo punto il programma utente è pronto per essere eseguito e può chiamare le procedure presenti nella DLL come se fossero staticamente legate al programma stesso.

L'alternativa a questo tipo di collegamento è, ovviamente, il **collegamento esplicito**. Questo approccio non richiede l'uso di librerie importate e non obbliga a caricare le DLL nello stesso momento in cui viene caricato il programma dell'utente. Al contrario il programma dell'utente effettua a tempo di esecuzione una chiamata esplicita per legare a sé una DLL e successivamente effettua altre chiamate per ottenere gli indirizzi delle procedure di cui ha bisogno. Dopo aver recuperato questi indirizzi il programma può chiamare le procedure. Quando ha finito di utilizzarle effettua una chiamata finale per scollegarsi dalla DLL. Quando anche l'ultimo processo si è scollegato da una DLL questa può essere rimossa dalla memoria.

È importante comprendere che, a differenza dei thread e dei processi, in una DLL le procedure non hanno una propria identità. Queste procedure vengono eseguite nel thread del chiamante e utilizzano lo stack del chiamante per le proprie variabili locali. Possono avere alcuni dati statici specifici di un processo (così come dati condivisi), ma si comportano esattamente come le procedure collegate staticamente. L'unica vera differenza riguarda il modo in cui viene effettuato il collegamento.

Collegamento dinamico in UNIX

Il sistema UNIX ha un meccanismo che è essenzialmente simile alle DLL di Windows e che si basa su quella che viene chiamata **libreria condivisa**. Analogamente ai file DLL una libreria condivisa è un file archivio contenente procedure o moduli di dati presenti in memoria a tempo di esecuzione e può essere collegata contemporaneamente a più processi. La libreria standard di C e gran parte del codice relativo alla comunicazione via rete sono esempi di librerie condivise.

UNIX supporta esclusivamente il collegamento implicito e quindi una libreria condivisa consiste in due parti: **statica** (collegata staticamente al file eseguibile) e **destinazione** (richiamata a tempo di esecuzione). Anche se alcuni dettagli presentano delle differenze, i concetti sono essenzialmente uguali a quelli delle DLL.

7.5 Riepilogo

Anche se la maggior parte dei programmi può, e dovrebbe, essere scritta in un linguaggio ad alto livello, in alcune situazioni è necessario utilizzare, almeno in parte, il linguaggio assemblativo. Candidati alla scrittura in linguaggio assemblativo sono i programmi per calcolatori portatili dotati di poche risorse, come le smart card, i processori integrati e i PDA wireless. Un programma in linguaggio assemblativo è una rappresentazione simbolica di un programma scritto nel linguaggio macchina di un livello inferiore. Esso viene tradotto nel linguaggio macchina mediante un programma chiamato assemblatore.

Per le applicazioni che richiedono un'alta velocità di esecuzione si sceglie di scrivere inizialmente l'intero programma in un linguaggio ad alto livello, e poi si determinano i punti in cui viene spesa la maggior parte del tempo; quindi si riscrivono in linguaggio assemblativo solo le porzioni di codice usate in modo più intenso. Questo approccio ha dei vantaggi rispetto a scrivere l'intero programma in codice assemblativo, dato che in genere solo una piccola frazione del codice è in realtà responsabile della maggior parte del tempo di esecuzione.

Molti assemblatori permettono l'uso delle macro, le quali consentono al programmatore di attribuire nomi simbolici a sequenze di codice usate frequentemente. In un secondo momento questi nomi vengono automaticamente sostituiti con il codice corrispondente. Di solito le macro possono anche essere parametrizzate in modo semplice. Le macro sono implementate mediante algoritmi di elaborazione di stringhe di caratteri.

La maggior parte degli assemblatori prevede due passate. La prima si occupa di costruire una tabella per le etichette, i letterali e gli identificatori dichiarati esplicitamente. I simboli possono essere memorizzati in modo disordinato e reperiti mediante ricerche lineari, oppure possono essere tenuti in ordine e poi individuati con una ricerca dicotomica. Un'altra possibilità per memorizzare e cercare i simboli consiste nell'uso di una tabella hash. Se i simboli non devono essere cancellati durante la prima passata, questo metodo solitamente risulta il migliore. La seconda passata genera il codice. Alcune pseudoistruzioni vengono eseguite durante la prima passata, altre durante la seconda.

I programmi assemblati indipendentemente possono essere collegati fra loro per formare un codice binario che può essere eseguito. Questo lavoro è realizzato dal linker. I suoi compiti principali sono la rilocazione e il collegamento dei nomi. Il collegamento dinamico è una tecnica grazie alla quale alcune procedure non sono collegate fino al momento in cui non vengono effettivamente richiamate. Le DLL di Windows e le librerie condivise di UNIX usano il collegamento dinamico.

Problemi

1. In un dato programma il 2% del codice è responsabile del 50% del tempo di esecuzione. Si confrontino le seguenti strategie relative ai tempi di programmazione e di esecuzione. Si assuma che il programma richieda 100 mesi/uomo di lavoro per essere programmato in C; la scrittura del codice assemblativo è invece 10 volte più lenta, ma il programma ottenuto risulta quattro volte più efficiente.
   a. intero programma in C
   b. intero programma in linguaggio assemblativo
   c. prima tutto in C, poi il 2% più critico riscritto in linguaggio assemblativo.
2. Le considerazioni che valgono per gli assemblatori a due passate, valgono anche per i compilatori?
   a. si assuma che i compilatori producano moduli oggetto, non codice assemblativo
   b. si assuma che i compilatori producano linguaggio assemblativo simbolico.
3. La maggior parte degli assemblatori per le CPU Intel ha l'indirizzo destinazione come primo operando e l'indirizzo sorgente come secondo operando. Quali problemi si sarebbero dovuti risolvere per utilizzare gli operandi in posizione invertita?
4. È possibile assemblare in due passate il seguente programma? `EQU` è una pseudoistruzione che mette in relazione l'etichetta con l'espressione che si trova nel campo dell'operando.

```
P EQU Q
Q EQU R
R EQU S
S EQU 4
```

5. La Società di Calcolatori DaDueSoldi sta pianificando di produrre un assemblatore per un calcolatore con parole a 48 bit. Per contenere i costi il responsabile del progetto, il Dott. Paperone, ha deciso di limitare la lunghezza degli identificatori in modo che possano essere memorizzati in una sola parola. Paperone ha dichiarato che gli identificatori possono consistere solamente in lettere, tranne la Q che è vietata (per superstizione: infatti è la 17ª lettera dell'alfabeto inglese). Qual è la massima lunghezza dei simboli? Si descriva un proprio schema di codifica.
6. Qual è la differenza tra istruzioni e pseudoistruzioni?
7. Qual è la differenza (se c'è) tra il contatore di locazioni delle istruzioni (ILC) e il contatore d'istruzioni (PC)? Dopo tutto entrambi tengono traccia della successiva istruzione in un programma.
8. Si mostri la tabella dei simboli dopo che si sono incontrate le seguenti istruzioni del Pentium 4. La prima istruzione è assegnata all'indirizzo 1000.

```
EVEREST:    POP BX          (1 BYTE)
K2:         PUSH BP         (1 BYTE)
WHITNEY:    MOV BP,SP       (2 BYTES)
MCKINLEY:   PUSHX           (3 BYTES)
FUJI:       PUSH SI         (1 BYTE)
KIBO:       SUB SI,300      (3 BYTES)
```

9. È possibile immaginare delle circostanze nelle quali il linguaggio assemblativo permetta che un'etichetta sia identica a un codice operativo (per esempio, *MOV* come etichetta)? Si motivi la risposta.

10. Si mostrino i passi necessari per cercare, utilizzando la ricerca dicotomica, la parola Berkley all'interno della seguente lista: Ann Arbor, Berkeley, Cambridge, Eugene, Madison, New Haven, Palo Alto, Pasadena, Santa Cruz, Stony Brook, Westwood e Yellow Springs. Quando si calcola l'elemento centrale di una lista con un numero pari di elementi si utilizzi l'elemento che si trova subito dopo il punto medio.
11. È possibile utilizzare la ricerca dicotomica su una tabella la cui dimensione è un numero primo?
12. Si calcoli il codice hash dei seguenti simboli sommando le lettere (A = 1, B = 2, ecc.) e calcolando il modulo tra il risultato e la dimensione della tabella hash. La tabella hash ha 19 posizioni, numerate da 0 a 18.

    els, jan, jelle, maaike

    Ciascuno di questi simboli genera un unico valore hash? In caso negativo, com'è possibile gestire il problema della collisione?
13. Il metodo della codifica hash descritto nel testo inserisce in una lista concatenata tutti gli elementi che hanno lo stesso codice hash. Un metodo alternativo è quello di avere un'unica tabella a n posizioni, in cui ciascuna ha lo spazio per memorizzare una chiave e il suo valore (o un puntatore a quest'ultimo). Se l'algoritmo di hash indica una posizione già occupata, si riprovi con un secondo algoritmo di hash. Se anche questa posizione è già occupata, se ne utilizzi un altro, continuando allo stesso modo finché non se ne trovi una vuota. Se la frazione di posizioni che sono occupate è R, quanti tentativi saranno necessari, in media, per poter inserire un nuovo simbolo?
14. Grazie ai progressi tecnologici un giorno potrebbe essere possibile collocare su un chip migliaia di CPU identiche, ciascuna delle quali con poche parole di memoria locale. Se tutte le CPU possono leggere e scrivere tre registri condivisi com'è possibile implementare una memoria associativa?
15. Il Pentium 4 ha un'architettura segmentata, composta da segmenti indipendenti. Un assemblatore per questa macchina potrebbe avere una pseudosistruzione `SEG N`, per indicare all'assemblatore di collocare il codice successivo nel segmento N. Questo schema avrebbe qualche influenza su ILC?
16. I programmi spesso si collegano a più DLL. Non sarebbe più efficiente inserire tutte le procedure in un'unica grande DLL e collegarla al programma?
17. È possibile mappare una DLL negli spazi degli indirizzi virtuali di due processi in due indirizzi virtuali diversi? Se sì, quali problemi sorgono? Possono essere risolti? Altrimenti, che cosa si può fare per eliminarli?
18. Un modo per effettuare il collegamento (statico) è il seguente. Prima di scandire la libreria il linker costruisce una lista delle procedure richieste, cioè una lista composta dai nomi definiti come `EXTERN` nei moduli che sta collegando. In seguito il linker analizza linearmente tutta la libreria, estraendo ogni procedura che si trova nella lista di quelle necessarie. Questo schema può funzionare? Altrimenti, perché no, e che rimedi si possono apportare?
19. È possibile utilizzare un registro come parametro attuale in una chiamata di macro? E una costante? Perché sì o perché no?
20. Si sta implementando un assemblatore con macro. Per ragioni estetiche il capo progetto ha deciso che le definizioni di macro non devono precedere le loro chiamate. Quali conseguenze ha questa decisione sull'implementazione?
21. Si pensi a un modo per far entrare un assemblatore con macro in un ciclo infinito.

22. Un linker legge cinque moduli, le cui lunghezze sono rispettivamente 200, 800, 600, 500 e 700. Se i moduli vengono caricati in quest'ordine quanto valgono le costanti di rilocazione?

23. Si scrivano due routine per gestire una tabella di simboli: *enter(symbol, value)* e *lookup(symbol, value)*. La prima inserisce i nuovi simboli nella tabella, mentre la seconda permette di cercare un simbolo al suo interno. Si utilizzi una codifica hash.

24. Si scriva un semplice assemblatore per il calcolatore Mic-1 del Capitolo 4. Si fornisca la possibilità di assegnare durante la fase di assemblaggio valori costanti ai simboli e un modo per assemblare una costante in una parola della macchina.

25. Si aggiunga la funzionalità delle macro all'assemblatore dell'esercizio precedente.

Capitolo 8

Architetture per il calcolo parallelo

Benché i calcolatori diventino sempre più veloci, le aspettative dei loro utenti crescono almeno altrettanto rapidamente. L'ambizione degli astronomi è la simulazione dell'intera storia dell'universo, dal big bang alla fine dei tempi. L'industria farmaceutica gradirebbe progettare sui propri computer medicine per la cura mirata di determinate malattie, invece di sacrificare intere legioni di ratti. I progettisti aeronautici sarebbero in grado di scoprire prodotti dotati di maggiore efficienza energetica se solo i calcolatori potessero farsi carico di tutto il lavoro, senza bisogno di costruire prototipi reali da valutare nella galleria del vento. È chiaro che, anche se è ormai disponibile molta potenza di calcolo, per un gran numero di utenti, e in special modo per gli scienziati e per l'industria, questa potenza non è mai abbastanza.

Nonostante il costante incremento delle frequenze di clock, la velocità dei circuiti non può aumentare indefinitamente. La velocità della luce costituisce già oggi un limite pratico per i progettisti dei calcolatori di fascia alta e le prospettive di riuscire a far muovere gli elettroni e i fotoni ancora più velocemente sono scarse. I problemi di dissipazione del calore fanno sì che i supercalcolatori diventino sempre più sofisticati condizionatori d'aria. Infine, le dimensioni dei transistor continueranno a diminuire e raggiungeranno presto il punto in cui conterranno così pochi atomi che gli effetti della meccanica quantistica (per esempio il principio di indeterminazione di Heisenberg) porranno seri problemi.

In conseguenza di tutto ciò, gli architetti degli elaboratori si rivolgono sempre più spesso al calcolo parallelo per trattare problemi di dimensioni sempre crescenti. Anche se non è possibile realizzare un calcolatore con una sola CPU e un ciclo di 0,001 ns, sarebbe sicuramente possibile costruirne uno con 1000 CPU, ciascuna con ciclo di clock di 1 ns. Sebbene il secondo progetto si avvalga di CPU più lente, in teoria la sua capacità totale di calcolo è la stessa del primo. Questa è l'idea cui sono affidate le speranze per il futuro.

Il parallelismo può essere introdotto a vari livelli. A quello più basso, può essere incorporato nel chip della CPU attraverso un progetto superscalare a pipeline con molteplici unità funzionali; oppure può essere introdotto con l'adozione d'istruzioni basate su parole molto lunghe e dotate di parallelismo implicito. La CPU può essere equipaggiata con caratteristiche speciali che le consentano di gestire il controllo di più thread alla volta, ed è anche possibile integrare più CPU sullo stesso chip. Tutti questi stratagemmi, se usati congiuntamente, possono produrre un incremento delle prestazioni per un massimo del 10% rispetto a un progetto puramente sequenziale.

A livello successivo, è possibile aggiungere al sistema delle schede dotate di CPU che arricchiscano le sue capacità di calcolo. In genere queste CPU *plug in* (cioè a innesto) svolgono funzioni specializzate, come l'elaborazione di pacchetti di rete, l'elaborazione multimediale o la crittografia. Consentono un fattore d'incremento delle prestazioni compreso tra 5 e 10, ma solo nel caso di applicazioni specializzate.

D'altra parte, l'unico modo per ottenere un incremento dell'ordine delle centinaia, delle migliaia o dei milioni, è di replicare il numero delle CPU, facendole cooperare in modo efficiente. Questa è l'idea alla base dei multiprocessori e dei multicomputer (*cluster di computer*). È evidente che il montaggio di migliaia di processori in un unico grande sistema costituisce di per sé un problema che va risolto.

Va detto infine che oggi è possibile strutturare su Internet intere organizzazioni di computer come reti di calcolo (griglie) con connessioni lasche. Sono sistemi alle prime armi, ma manifestano un grosso potenziale per il futuro.

Due CPU o elementi di calcolo contigui che svolgono calcoli in modo molto interattivo e la cui connessione ha un'elevata larghezza di banda e un ritardo trascurabile, si dicono **legati strettamente** (*tightly coupled*). Viceversa si dicono **legati debolmente** (*loosely coupled*) se si trovano molto distanti, hanno una banda ridotta, un ritardo elevato e svolgono i loro calcoli in modo poco interattivo. In questo capitolo ci interessiamo ai principi progettuali delle varie forme di parallelismo e ne analizziamo una varietà di esempi. Cominciamo dai sistemi a legame più stretto, cioè quelli che usano il parallelismo nel chip, spostandoci gradualmente verso sistemi con legame più debole, per finire con lo spendere qualche parola sul *grid computing* ("calcolo su griglie di computer"). La Figura 8.1 mostra lo spettro delle possibilità (da legame più stretto a più debole).

La questione del parallelismo è oggetto di intensa ricerca a tutti i livelli dello spettro. Perciò nel corso del capitolo segnaleremo un gran numero di riferimenti bibliografici, rivolgendo particolare attenzione agli articoli più recenti. I riferimenti di natura più introduttiva sono elencati nel Paragrafo 9.1.8.

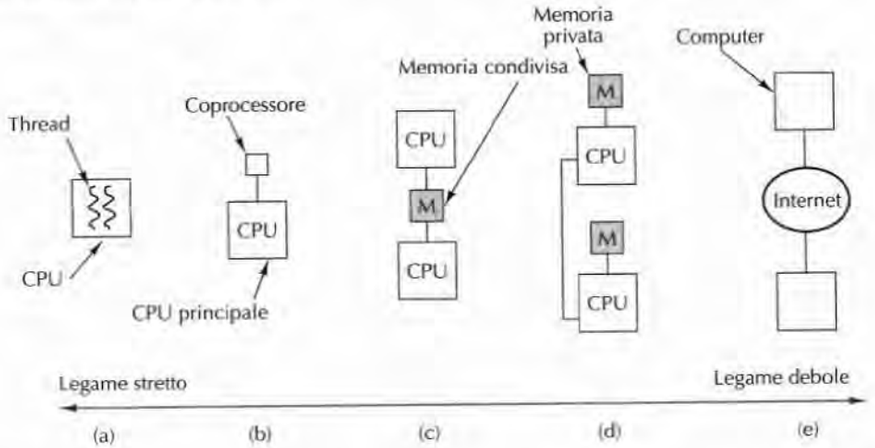


Figura 8.1 (a) Parallelismo nel chip. (b) Coprocessore. (c) Multiprocessore. (d) Multicomputer. (e) Griglia.

8.1 Parallelismo nel chip

Un modo per incrementare la produttività di un chip è di fargli svolgere più compiti alla volta. In questo paragrafo ci interessiamo ai modi in cui si può ottenere un incremento delle prestazioni tramite il parallelismo a livello del chip, compreso il parallelismo del livello delle istruzioni, il multithreading e la coabitazione di più CPU sullo stesso chip. Sono tecniche abbastanza diverse e ciascuna dà un proprio contributo. Il loro comune obiettivo è quello di svolgere più attività nello stesso tempo.

8.1.1 Parallelismo a livello delle istruzioni

A livello delle istruzioni, un modo per ottenere il parallelismo è di emettere più istruzioni per ciclo di clock. Ci sono due tipi di CPU a emissione multipla: processori superscalari e processori VLIW. Vi abbiamo già accennato precedentemente, ma può essere utile ritornare sull'argomento.

Abbiamo già visto le CPU superscalari (Figura 2.5). Nel caso più generale, c'è un momento lungo la pipeline in cui un'istruzione è pronta per essere eseguita. Le CPU superscalari sono in grado di emettere, verso le unità di esecuzione, più istruzioni in un solo ciclo. Il numero d'istruzioni emesse dipende dal progetto del processore e dalle circostanze contingenti. L'hardware determina il numero massimo d'istruzioni che può essere emesso, in genere da due a sei. In ogni caso un'istruzione non verrà emessa se attende un calcolo che non è stato ancora completato o se necessita di un'unità funzionale non disponibile.

L'altra forma di parallelismo a livello delle istruzioni è quello dei processori **VLIW** (*Very Long Instruction Word*, "con parola d'istruzione molto lunga"). Nella loro forma originaria, le macchine VLIW avevano davvero parole molto lunghe per contenere istruzioni che usavano diverse unità funzionali. Si consideri per esempio la Figura 8.2(a), in cui la macchina dispone di cinque unità funzionali e può eseguire simultaneamente due operazioni intere, una in virgola mobile, un caricamento e una memorizzazione. Un'istruzione VLIW per questa macchina conterrebbe cinque codici operativi e cinque coppie di operandi: un codice e una coppia di operandi per ogni unità funzionale. Se ci sono 6 bit per codice operativo, 5 bit per registro e 32 bit per gli indirizzi di memoria, le istruzioni possono raggiungere facilmente i 134 bit, il che fa di loro istruzioni abbastanza lunghe.

Questo progetto si rivelò troppo rigido perché non tutte le istruzioni riuscivano a sfruttare tutte le unità funzionali, il che conduceva all'uso di molte inutili `NO-OP`, come illustrato nella Figura 8.2(b). Perciò le macchine VLIW moderne dispongono di una modalità di costruzione di pacchetti (*bundle*) d'istruzioni, conclusi per esempio dal bit di "fine pacchetto", come mostrato nella Figura 8.2(c). Il processore è in grado di effettuare il fetch e l'emissione di un intero pacchetto per volta. Spetta al compilatore preparare pacchetti d'istruzioni compatibili.

A tutti gli effetti, questa tecnica sposta dall'esecuzione alla compilazione il difficile compito di stabilire quali istruzioni possono essere avviate alla esecuzione contemporanea. Questa scelta consente di avere un hardware più semplice e veloce e, poiché un compilatore può far durare l'ottimizzazione per tutto il tempo necessario, permette di assemblare pacchetti d'istruzioni migliori di quanto potrebbe fare l'hardware durante l'esecuzione. Per contro, questo progetto comporta un cambio radicale nelle architetture delle CPU che sarà difficile da far accettare, come dimostrato dalla scarsa accoglienza suscitata da Itanium.

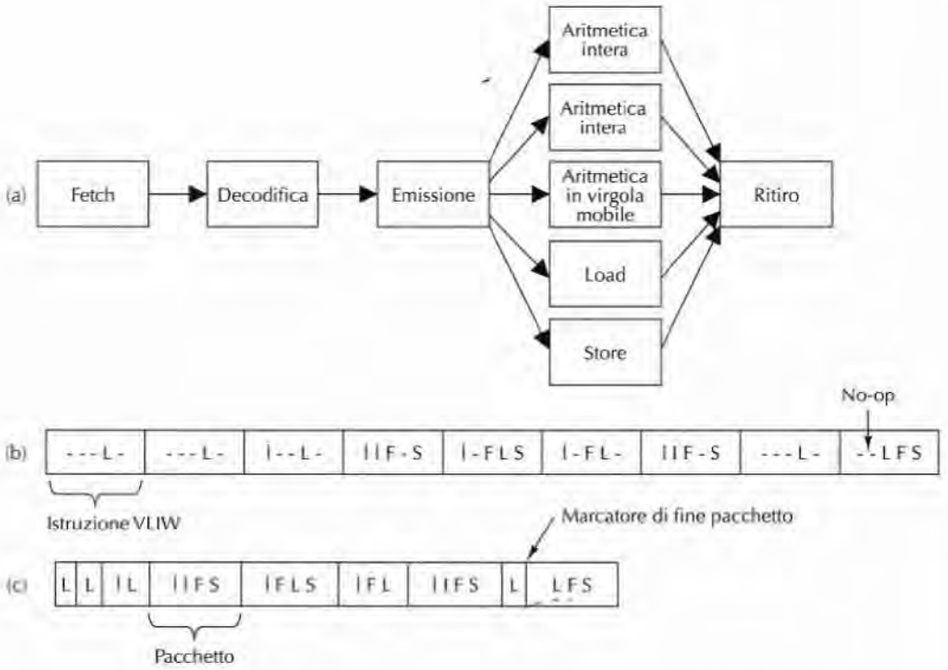


Figura 8.2 (a) Pipeline della CPU. (b) Sequenza d'istruzioni VLIW. (c) Flusso d'istruzioni con marcatori di pacchetto.

Val la pena notare qui che il parallelismo a livello delle istruzioni non è la sola forma di parallelismo di basso livello. C'è infatti il parallelismo a livello della memoria, secondo cui diverse operazioni di memoria vengono svolte contemporaneamente (Chou et al., 2004).

CPU VLIW TriMedia

Nel Capitolo 5 abbiamo studiato l'Itanium-2, un esempio di CPU VLIW. Analizziamo ora **TriMedia**, un processore VLIW molto diverso e progettato da Philips, la compagnia olandese che ha anche inventato i CD audio e i CD-ROM. TriMedia è stato pensato come processore integrato per applicazioni intensive di tipo audio, video o su immagini, nei lettori CD, DVD o Mp3, nei registratori CD o DVD, nelle console TV interattive, nelle macchine fotografiche e videocamere digitali, e così via. Proprio perché destinato a questo tipo di applicazioni, non sorprende che TriMedia differisca molto da Itanium-2, che è invece una CPU per uso generale concepita per i server di fascia alta.

TriMedia è un vero processore VLIW in cui ogni istruzione può contenere cinque **operazioni**. In condizioni ideali, a ogni ciclo di clock viene avviata l'esecuzione di un'istruzione e l'emissione delle cinque operazioni. Il clock ha una frequenza di 266 o 300 MHz, che però equivale a una frequenza cinque volte maggiore, visto che può eseguire cinque operazioni per ciclo. In seguito ci concentreremo sull'implementazione TM3260 di TriMedia (esistono anche altre versioni, le cui differenze sono però trascurabili).

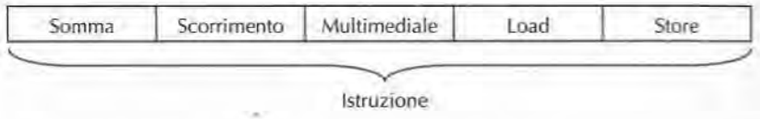


Figura 8.3 Tipica istruzione TriMedia, con cinque operazioni possibili.

La Figura 8.3 illustra un'istruzione tipica. Si va dalle istruzioni intere standard di 8, 16 e 32 bit, alle istruzioni in virgola mobile IEEE 754, fino alle istruzioni multimediali parallele. Grazie all'emissione di cinque istruzioni multimediali per ciclo, TriMedia è sufficientemente veloce da decodificare un flusso di dati proveniente da una videocamera senza ridurre la dimensione delle immagini, né la frequenza di acquisizione.

TriMedia ha una memoria orientata al byte, con i registri di I/O mappati nello spazio di memoria. Le mezze parole (16 bit) e le parole intere (32 bit) devono essere allineate lungo le loro estremità (ovvero devono iniziare in byte di indirizzo pari o – per le parole intere – multiplo di 4). L'ordinamento dei byte può essere big-endian o little-endian, a seconda di un bit di `PSW` che può essere impostato dal sistema operativo. Questo bit ha effetto solo sul modo in cui operano i trasferimenti di caricamento e memorizzazione tra i registri e la memoria. La CPU contiene una cache specializzata associativa a 8 vie, con linee di 64 byte sia per la cache delle istruzioni, sia per quella dei dati. La cache delle istruzioni è di 64 KB, quella dei dati è di 16 KB.

Ci sono 128 registri di 32 bit per uso generale. Il registro `R0` è cablato al valore 0, `R1` a 1. I tentativi di modificare uno di questi registri hanno sulla CPU l'effetto di un infarto. I restanti 126 registri sono tutti equivalenti dal punto di vista funzionale e possono essere usati per qualsiasi scopo. Inoltre esistono quattro registri di 32 bit specializzati: il program counter, `PSW` e due registri legati agli interrupt. Infine, viene utilizzato un registro di 64 bit per contare i cicli dall'ultimo reset della CPU. A 300 MHz, ci vogliono circa 2000 anni perché il contatore vada in overflow.

Il TriMedia TM3260 ha 11 diverse unità funzionali per svolgere le operazioni aritmetiche, logiche e di controllo del flusso, oltre a una per il controllo di cache che non prenderemo in considerazione. Le prime due colonne della Figura 8.4 indicano il nome dell'unità e ne danno una breve descrizione, la terza specifica il numero di copie hardware dell'unità e la quarta dà la latenza dell'unità, cioè il numero di cicli di clock che impiega per completare l'operazione. Notiamo qui che tutte le unità funzionali sono a pipeline, eccetto la unità VM (in virgola mobile) per la divisione/radice quadrata. La latenza indicata in tabella stabilisce il tempo necessario alla disponibilità del risultato di un'operazione, ma a ogni ciclo è possibile iniziare una nuova operazione. Per esempio, tre istruzioni consecutive possono contenere ciascuna due caricamenti, così in un certo istante saranno in esecuzione sei operazioni di caricamento, ma a stadi differenti.

Infine, le ultime cinque colonne mostrano le possibili corrispondenze tra posizione delle operazioni nell'istruzione e le unità funzionali. Per esempio, l'operazione di confronto in virgola mobile deve occupare necessariamente il terzo posto di un'istruzione.

L'unità costante si usa per le operazioni immediate, come il caricamento in un registro di un numero contenuto nell'operazione stessa. L'ALU intera svolge la somma, la sottrazione e le usuali operazioni booleane, e inoltre impacchetta/spacchetta le operazioni. L'unità di scorrimento può traslare il contenuto di un registro di un certo numero di bit in entrambe le direzioni.

Unità	Descrizione	#	Latenza	1	2	3	4	5
Costante	Operazioni immediate	5	1	x	x	x	x	x
ALU intera	Aritmetica a 32 bit, operazioni booleane	5	1	x	x	x	x	x
Scorrimento	Scorrimento di più bit	2	1	x	x	x	x	x
Load/Store	Operazioni di memoria	2	3				x	x
MUL Int/VM	Moltiplicazione intera e VM a 32 bit	2	3		x	x		
ALU VM	Aritmetica VM	2	3	x			x	
Confronto VM	Confronti VM	1	1			x		
sqrt/div VM	Divisione e radice quadrata VM	1	17		x			
Salti	Controllo del flusso	3	3		x	x	x	
ALU DSP	Aritmetica multimediale a due o quattro operandi (16 o 8 bit)	2	3	x		x		x
MUL DSP	Moltiplicazioni multimediali a due o quattro operandi (16 o 8 bit)	2	3	x		x		

Figura 8.4 Unità funzionali di TM3260; numero, latenza e posizioni che possono avere nelle istruzioni.

L'unità load/store effettua il fetch delle parole di memoria verso i registri o viceversa. TriMedia è essenzialmente una CPU RISC potenziata, e le operazioni normali lavorano sui registri, mentre soltanto l'unità load/store è usata per gli accessi alla memoria, con trasferimenti di 8, 16 o 32 bit.

L'unità per la moltiplicazione gestisce sia i prodotti tra interi, sia tra numeri in virgola mobile. Le tre unità successive gestiscono la somma/sottrazione in virgola mobile, i confronti, la radice quadrata e la divisione.

Le operazioni di salto sono eseguite dall'unità di salto. Dopo un salto c'è un ritardo di 3 cicli, perciò le tre istruzioni successive a un salto (fino a 15 operazioni) vengono eseguite sempre, anche nel caso di salti incondizionati.

Veniamo infine alle unità multimediali per la gestione delle speciali operazioni multimediali. Il termine DSP sta per *Digital Signal Processor* ("processore dei segnali digitali"), le cui funzioni sono svolte in questo caso proprio dalle operazioni multimediali. Prima di entrare nei dettagli di queste operazioni, segnaliamo che usano tutte l'**aritmetica satura** (*saturated arithmetic*) invece dell'aritmetica in complemento a due usata dalle operazioni intere. Quando un'operazione produce un risultato che non può essere espresso a causa di un overflow, non viene sollevata un'eccezione, né viene restituito un risultato inservibile, bensì si ottiene il numero valido più vicino a quello corretto. Per esempio, se si usano i numeri di 8 bit senza segno, la somma di 130 + 130 dà 255.

A causa del fatto che non tutte le operazioni possono occupare tutti i posti, accade frequentemente che un'istruzione non contenga tutte e cinque le operazioni che può ospitare. Se un posto non è utilizzato, la sua dimensione viene ridotta per minimizzare lo spreco di spazio. Le operazioni presenti occupano 26, 34 o 42 bit. A seconda delle operazioni presenti, un'istruzione TriMedia può essere lunga dai 2 ai 28 byte, compresa l'intestazione di dimensione costante.

TriMedia non effettua controlli in fase d'esecuzione per verificare che le operazioni in un'istruzione siano compatibili. Se non lo sono, vengono eseguite comunque e producono un risultato errato. L'assenza del controllo è stata una scelta progettuale deliberata e intesa a risparmiare tempo e transistor. Il Pentium, durante l'esecuzione, svolge i controlli necessari per sincerarsi della compatibilità di tutte le operazioni superscalari, ma paga questa certezza in complessità, tempo e transistor. TriMedia si risparmia questa spesa affidando lo scheduling al compilatore, che dispone di tutto il tempo che vuole per ottimizzare con cura il posizionamento delle operazioni nelle istruzioni. Il rovescio della medaglia è che, se un'operazione necessita di un'unità funzionale non disponibile, l'intera istruzione va in stallo finché l'unità non torna disponibile.

Come nell'Itanium-2, le operazioni TriMedia sono predicative. Ogni operazione (a parte due eccezioni trascurabili) specificano un registro da esaminare prima dell'esecuzione dell'operazione; se il bit meno significativo del registro è asserito, l'istruzione viene eseguita, viceversa viene saltata. A ognuna delle (al massimo) cinque operazioni viene attribuito un predicato individuale. Un esempio di operazione predicativa è

```
IF R2 IADD R4, R5 -> R8
```

che effettua il test su `R2` e, se il suo bit meno significativo vale 1, somma `R4` a `R5` e salva il risultato in `R8`. Un'operazione diventa incondizionata se usa `R1` (che vale sempre 1) come registro predicativo. Se usa `R0` (che vale sempre 0) diventa una no-op.

Le operazioni multimediali di TriMedia possono essere ripartite nei 15 gruppi della Figura 8.5. Molte operazioni coinvolgono tagli (*clipping*) mediante la specifica di un operando e di un intervallo; il taglio forza l'operando all'interno dell'intervallo, ovvero se il suo valore cade fuori dall'intervallo gli attribuisce il valore di uno degli estremi. Possono essere tagliati operandi di 6, 16 o 32 bit. Per fare un esempio, i tagli di 40 e di 340 in un intervallo da 0 a 255 restituiscono rispettivamente 40 e 255. Il gruppo dei tagli effettua appunto operazioni di taglio.

I quattro gruppi successivi della Figura 8.5 eseguono le operazioni indicate su operandi di varie dimensioni, tagliando il risultato in un certo intervallo. Il gruppo min, max esamina due registri e, per ogni byte, trova il valore minimo o massimo. Allo stesso modo, il gruppo dei confronti considera due registri come quattro coppie di byte ed effettua i confronti su ogni coppia.

Le operazioni multimediali non sono eseguite quasi mai su interi di 32 bit perché le immagini sono fatte di pixel RGB, definiti da 8 pixel per ciascuno dei colori rosso, verde e blu. Durante l'elaborazione di un'immagine (per esempio la sua compressione) questa viene rappresentata in genere da tre componenti, uno per ogni colore (spazio RGB) o in una forma logicamente equivalente (spazio YUV, descritto nel seguito del capitolo). Comunque sia, gran parte dell'elaborazione si svolge su array rettangolari contenenti interi di 8 bit senza segno.

TriMedia è dotato di molte operazioni ideate specificatamente per l'elaborazione efficiente di interi senza segno di 8 bit. Consideriamo il semplice esempio del vertice superiore sinistro di un array di valori di 8 bit, collocati in memoria (*big-endian*) della Figura 8.6(a). I 4 × 4 blocchi del vertice contengono 16 valori di 8 bit, etichettati da *A* a *P*. Se si vuole trasporre l'immagine, per produrre il risultato della Figura 8.6(b), come si può fare?

Gruppo	Descrizione
Tagli	Tagli di 4 byte o di 2 mezze parole
Valore assoluto DSP	Valore assoluto saturato con segno
Somma DSP	Somma saturata con segno
Sottrazione DSP	Sottrazione saturata con segno
Moltiplicazione DSP	Moltiplicazione saturata con segno
Min, max	Minimo o massimo di quattro coppie di byte
Confronti	Confronti byte per byte di due registri
Scorrimento	Scorrimento di una coppia di operandi di 16 bit
Somma dei prodotti	Somma con segno di prodotti di 8 o 16 bit
Fusione, impacchettamento, scambio	Manipolazione di byte o mezze parole
Media di quattro byte	Media senza segno di quattro operandi, byte per byte
Media di byte	Media senza segno di quattro elementi, byte per byte
Moltiplicazione di byte	Moltiplicazione senza segno a 8 bit
Stima del movimento	Somma senza segno di valori assoluti di differenze di 8 bit con segno
Miscellanea	Altre operazioni aritmetiche

Figura 8.5 Principali gruppi di operazioni di TriMedia.

Una possibilità prevede l'utilizzo di 12 operazioni per caricare i byte nei registri, seguite da 12 operazioni per memorizzare i byte nella loro posizione corretta (i quattro byte lungo la diagonale non si spostano durante la trasposizione). Il problema è che questo approccio richiede 24 operazioni (lunghe e lente) che accedono alla memoria.

In alternativa si può cominciare con quattro operazioni di caricamento di una parola per volta nei quattro registri da `R2` a `R5`, come mostrato nella Figura 8.6(c). Poi si ottengono le quattro parole di output della Figura 8.6(d) tramite operazioni di mascheratura e scorrimento. Infine, le parole sono pronte per essere salvate in memoria. Sebbene questo metodo riduca il numero di accessi in memoria da 24 a 8, le mascherature e degli scorrimenti sono costosi a causa del gran numero di operazioni richieste per estrarre e inserire ogni byte nella sua posizione corretta.

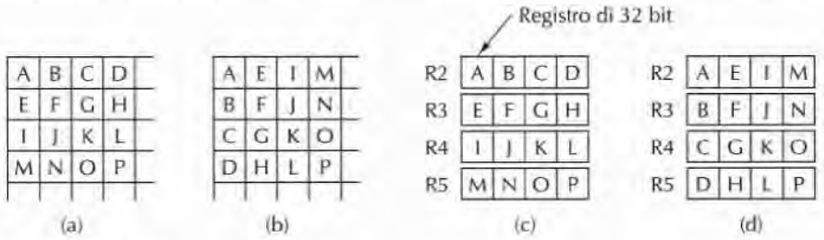


Figura 8.6 (a) Un array di elementi di 8 bit. (b) L'array trasposto. (c) L'array originale memorizzato in quattro registri. (d) L'array trasposto nei quattro registri.

TriMedia consente di raggiungere una soluzione migliore delle precedenti. Si comincia con il fetch delle quattro parole nei registri, però, per costruire l'output, invece delle mascherature e degli scorrimenti, si usano operazioni speciali per l'estrazione e l'inserzione di byte nei registri. In definitiva, la trasposizione può essere ottenuta con 8 accessi alla memoria e 8 operazioni multimediali speciali. Il codice comincia con un'istruzione contenente due `load` in posizione 4 e 5, per il caricamento nei registri `R2` e `R3`, seguita da un'altra istruzione per caricare `R4` e `R5`. Queste due istruzioni possono usare le altre posizioni per altri scopi. Al termine di tutti i caricamenti è possibile impacchettare le otto operazioni speciali per la costruzione dell'output in due istruzioni, seguite da altre due istruzioni per la memorizzazione. In totale servono solo sei istruzioni, in cui 14 delle 30 posizioni complessive sono ancora disponibili per ospitare altre operazioni. In effetti, il compito può essere svolto complessivamente con l'equivalente di circa tre istruzioni. Le altre operazioni multimediali sono altrettanto efficienti. Grazie alle sue potenti operazioni e alla capienza delle sue istruzioni, TriMedia è molto efficiente per la tipologia di calcolo richiesta dall'elaborazione multimediale.

8.1.2 Multithreading nel chip

Tutte le moderne CPU a pipeline presentano un problema: quando un riferimento in memoria fallisce nelle cache di primo e di secondo livello, bisogna aspettare molto tempo prima che la parola richiesta (e la corrispondente linea di cache) sia caricata nella cache, e così nel frattempo la pipeline è in stallo. Il **multithreading nel chip** costituisce un modo per trattare questa situazione perché, in una certa misura, si riescono a mascherare questi stalli consentendo alla CPU di gestire contemporaneamente più thread di controllo. Infatti, se il thread 1 è bloccato, la CPU ha ancora la possibilità di eseguire il thread 2 e di mantenere l'hardware impegnato.

Per quanto si tratti di un'idea molto semplice, ne esistono diverse varianti che ci accingiamo a esaminare. Il primo approccio si chiama **multithreading a grana fine** (*fine-grained multithreading*) ed è illustrato nella Figura 8.7, che mostra una CPU capace di emettere un'istruzione per ciclo di clock. Nelle Figure 8.7(a)-(c) osserviamo l'attività dei thread *A*, *B* e *C*, lungo 12 cicli di macchina. Durante il primo ciclo, *A* esegue l'istruzione *A1*, che viene completata in un ciclo, quindi nel secondo comincia l'esecuzione di *A2*. Sfortunatamente questa istruzione provoca un fallimento nella cache di primo livello e così si perdono due cicli per recuperare l'istruzione dalla cache di secondo livello. Il thread riprende dal ciclo 5. Anche

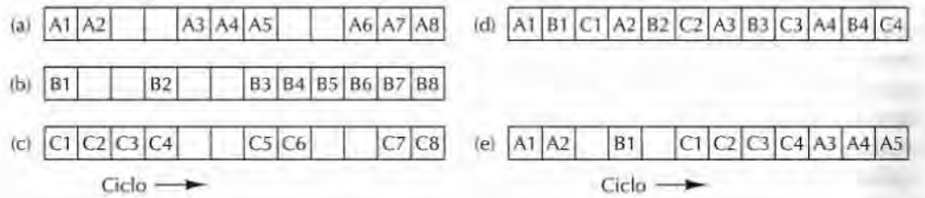


Figura 8.7 (a)-(c) Tre thread: le caselle vuote indicano che il thread è in stallo perché in attesa di dati dalla memoria. (d) Multithreading a grana fine. (e) Multithreading a grana grossa.

i thread *B* e *C* di tanto in tanto vanno in stallo, come illustrato nella figura. Secondo questo modello, quando un'istruzione va in stallo, le istruzioni successive non possono essere emesse. Naturalmente, alle volte è tuttavia possibile emettere altre istruzioni se si dispone di uno scoreboard più sofisticato, ma qui ignoriamo questa evenienza.

Il multithreading a grana fine nasconde gli stalli grazie all'esecuzione a turno dei thread, con una commutazione a ogni ciclo, come mostra la Figura 8.7(d). Quando arriva il quarto ciclo l'operazione di memoria avviata da *A1* è ormai stata completata e l'istruzione *A2* può essere eseguita, anche se richiede il risultato di *A1*. Nell'esempio ogni stallo dura al massimo due cicli, perciò le operazioni vengono sempre completate in tempo visto che ci sono tre thread. Se ci fossero stalli di tre cicli, avremmo bisogno di quattro thread per assicurare la continuità dell'attività, e così via.

Dal momento che non vi è alcuna relazione tra i thread, ciascuno ha bisogno del proprio insieme di registri. All'emissione di un'istruzione è necessario accludere all'istruzione stessa un puntatore al suo insieme di registri così che, se viene referenziato un registro, l'hardware possa sapere quale registro usare. Per questa ragione, il numero massimo di thread che può essere eseguito in parallelo è stabilito a priori in fase di progettazione del chip.

Le operazioni di memoria non sono l'unica causa di stallo. A volte un'istruzione aspetta il risultato di un'istruzione precedente non ancora completata, altre volte un'istruzione non può essere avviata perché segue un salto condizionato la cui destinazione non è ancora nota. Come regola generale, se la pipeline ha *k* stadi e ci sono almeno *k* thread in esecuzione a turno, nella pipeline non ci sarà mai più di un'istruzione per thread in esecuzione e perciò non si possono verificare conflitti. In queste condizioni la CPU può girare a pieno ritmo senza stalli.

Ovviamente potrebbero non esserci tanti thread quanti sono gli stadi della pipeline, perciò alcuni progettisti preferiscono un approccio differente, detto **multithreading a grana grossa** (*coarse-grained multithreading*), illustrato nella Figura 8.7(e). In questo caso *A* si avvia e continua a emettere istruzioni finché non va in stallo, causando lo spreco di un ciclo. A quel punto l'esecuzione viene commutata su *B1* ma, poiché la prima istruzione di *B* va subito in stallo, si verifica un'altra commutazione di thread e al ciclo 6 va in esecuzione *C1*. Visto che si perde un ciclo a ogni stallo, il multithreading a grana grossa è potenzialmente meno efficiente di quello a grana fine, ma presenta il vantaggio di richiedere meno thread per mantenere la CPU occupata. Il multithreading a grana grossa è preferibile in tutte quelle situazioni in cui c'è un numero insufficiente di thread attivi, perché garantisce di trovarne almeno uno da mandare in esecuzione.

Anche se abbiamo descritto il multithreading a grana grossa come una commutazione sugli stalli, non è questa l'unica alternativa. Un'altra possibilità è di effettuare la commutazione immediatamente tutte le volte in cui un'istruzione potrebbe causare uno stallo: si pensi ai caricamenti, alle memorizzazioni e ai salti, ancor prima di scoprire se lo stallo si verificherà davvero. Secondo questa strategia la commutazione occorre in anticipo (non appena l'istruzione viene decodificata) e potrebbe così far evitare i cicli morti. È un po' come dire: "andiamo avanti finché potremmo avere un problema, al che commutiamo giusto in tempo". Così facendo, il multithreading a grana grossa, con le sue frequenti commutazioni, somiglia un po' di più a quello a grana fine.

Indipendentemente dal tipo di multithreading usato, è necessario mantenere traccia dell'appartenenza delle operazioni ai thread. Nel caso del multithreading a grana fine l'unica

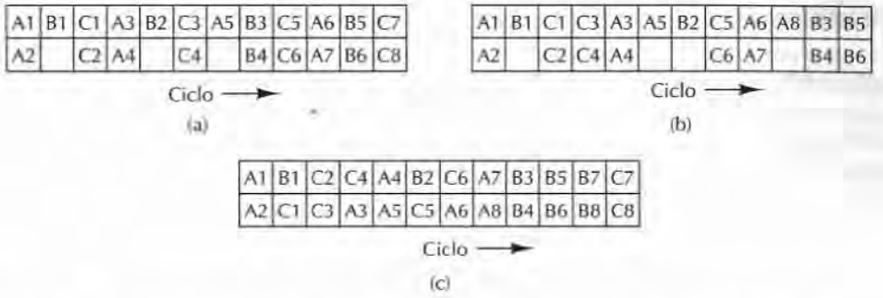


Figura 8.8 Multithreading in una CPU superscalare a doppia emissione. (a) Multithreading a grana fine. (b) Multithreading a grana grossa. (c) Multithreading simultaneo.

possibilità ragionevole è l'inserimento di un identificatore di thread in ogni operazione, così da poter rintracciare la sua identità mentre attraversa i diversi stadi della pipeline. Nel caso del multithreading a grana grossa si può far di meglio: svuotare la pipeline a ogni commutazione di thread. In tal modo c'è sempre un solo thread nella pipeline e così la sua identità non è mai messa in dubbio. È evidente che questa soluzione ha senso solo se il tempo che intercorre tra due commutazioni è molto più lungo del tempo di svuotamento della pipeline.

Fin qui abbiamo presupposto che la CPU possa emettere una sola istruzione per ciclo, ma abbiamo già visto che le CPU moderne ne possono emettere di più. Nella Figura 8.8 supponiamo che la CPU possa emettere due istruzioni per ciclo, pur conservando la regola che, se un'istruzione va in stallo, le successive non possono essere emesse. La Figura 8.8(a) mostra il funzionamento di una CPU superscalare a doppia emissione e con multithreading a grana fine. Le prime due istruzioni del thread *A* possono essere emesse nel primo ciclo, ma nel caso di *B* incontriamo subito un problema nel ciclo successivo, e così può essere emessa una sola istruzione, e così via.

La Figura 8.8(b) mostra il funzionamento di una CPU a doppia emissione con multithreading a grana fine, ma questa volta con uno schedulatore statico che non introduce un ciclo morto dopo lo stallo di un'istruzione. Praticamente, i thread si succedono a turno e la CPU emette due istruzioni per ogni thread finché non incontra uno stallo, nel qual caso commuta al thread successivo all'inizio del ciclo seguente.

C'è una terza possibilità di multithreading con le CPU superscalari, il **multithreading simultaneo**, illustrato nella Figura 8.8(c). Lo si può considerare un raffinamento del multithreading a grana grossa, in cui ciascun thread emette due istruzioni per ciclo fintanto che può, altrimenti, non appena raggiunge uno stallo, viene emessa immediatamente un'istruzione del thread che segue affinché la CPU resti pienamente impegnata. Il multithreading simultaneo aiuta anche a mantenere occupate le unità funzionali. Quando un'istruzione non può essere avviata perché necessita di un'unità funzionale occupata, si può scegliere al suo posto un'istruzione di un altro thread. Nella figura supponiamo che *B8* vada in stallo al ciclo 11, così *C7* viene avviata al ciclo 12.

Per maggiori informazioni sul multithreading si faccia riferimento a (Dean, 2004; Kalla et al., 2004; Kapil et al., 2004). (Sohi e Roth, 2001) hanno studiato la combinazione di multithreading ed esecuzione speculativa.

Hyperthreading nel Pentium 4

Lo studio astratto del multithreading ci consente ora di affrontare a un esempio pratico: quello del Pentium 4. Quando questo chip era già in produzione, gli architetti di Intel cominciarono a riflettere sul modo di velocizzare il processore senza modificare l'interfaccia con il programmatore, la qual cosa sarebbe stata inaccettabile. Furono individuate subito cinque ipotesi.

1. Incrementare la velocità di clock.
2. Posizionare due CPU nel chip.
3. Aggiungere unità funzionali.
4. Allungare la pipeline.
5. Usare il multithreading.

Un modo ovvio per migliorare le prestazioni consiste nell'aumento della velocità del clock senza cambiare nient'altro. È una soluzione relativamente semplice e ben fondata, perciò ogni nuovo chip in genere supera di poco la velocità del suo predecessore. Sfortunatamente, l'incremento delle velocità di clock porta con sé due svantaggi che ne limitano la tollerabilità. Un clock più veloce assorbe più energia, un problema molto grande per i computer portatili e in genere per tutti quelli alimentati a batteria. In secondo luogo, più energia in ingresso vuol dire più calore da dissipare.

La sistemazione di due CPU in un chip è abbastanza semplice, ma equivale circa a raddoppiare l'area del chip se ciascun processore è dotato di cache propria; la riduzione del numero di chip per unità di superficie di un fattore due non fa che raddoppiare i costi unitari di produzione. Se i due chip condividono una cache comune grande quanto quella originale, la superficie del chip non viene raddoppiata, ma la quantità di cache per CPU è dimezzata e ciò riduce le prestazioni. Inoltre, se le applicazioni dei server di fascia alta sono spesso in grado di trarre il massimo giovamento dalla presenza di più CPU, le applicazioni dei desktop non hanno abbastanza parallelismo da giustificarne due.

Anche l'aggiunta di unità funzionali è relativamente semplice, ma bisogna raggiungere il giusto equilibrio. È inutile disporre di 10 ALU se il chip non è in grado di passare loro le istruzioni abbastanza velocemente da tenerle occupate.

L'allungamento della pipeline tramite aggiunta di nuovi stadi, ciascuno destinato allo svolgimento di un compito più piccolo in un tempo più breve, può incrementare le prestazioni, ma aumenta anche gli effetti negativi delle predizioni errate sui salti, dei fallimenti di cache, degli interrupt e in genere di tutti quei fattori che interrompono il normale flusso normale esecutivo. Inoltre, per poter sfruttare pienamente la maggiore lunghezza della pipeline, è necessario un clock più veloce, il che implica un più elevato consumo di energia e un incremento di produzione di calore.

Resta infine l'introduzione del multithreading, il cui valore aggiunto sta nel permettere che un secondo thread utilizzi l'hardware che sarebbe rimasto altrimenti inattivo. A seguito di alcuni esperimenti, si è stabilito che il supporto del multithreading, a fronte di un incremento del 5% della superficie del chip, produce un miglioramento delle prestazioni del 25% per molte applicazioni ed è perciò una buona scelta. La prima CPU Intel a usare il multithreading è stata Xeon nel 2002; il multithreading è stato aggiunto in seguito al

Pentium 4 sin dalla versione a 3,06 GHz. Intel ha dato il nome di **hyperthreading** all'implementazione del multithreading nel Pentium 4.

Alla sua base c'è l'idea di consentire a due thread (o anche a due processi, dal momento che la CPU non può distinguere tra thread e processi) di essere eseguiti contemporaneamente. Dal punto di vista del sistema operativo, un Pentium 4 con hyperthreading somiglia a un biprocessore in cui le CPU condividono cache e memoria principale. Il sistema operativo seleziona i thread per l'esecuzione in modo indipendente; se ci sono due applicazioni in esecuzione nello stesso istante, il sistema operativo può farle eseguire in parallelo. Per esempio, se un demone di posta riceve o spedisce email in background[1] mentre un utente interagisce con un altro programma, i due programmi possono essere eseguiti contemporaneamente proprio come se ci fossero due CPU disponibili.

Il software applicativo progettato per l'esecuzione con thread multipli può avvalersi di entrambe le CPU virtuali. Per esempio, i programmi di montaggio video in genere permettono all'utente di specificare alcuni filtri da applicare ai fotogrammi di un certo intervallo. Questi filtri possono agire su luminosità, contrasto, bilanciamento del bianco e altre proprietà di ciascun fotogramma. Il programma può decidere di affidare i fotogrammi pari a una CPU e quelli dispari all'altra, e i due processori virtuali possono proseguire in modo del tutto indipendente.

Poiché i thread condividono tutte le risorse hardware, si rende necessaria una strategia per la gestione della condivisione. Intel ha identificato quattro strategie utili per la condivisione di risorse associata all'hypethreading: duplicazione e ripartizione di risorse, condivisione a soglia oppure totale. Vediamole una per volta.

Cominciamo dalle risorse che vengono duplicate per i thread. Per esempio, poiché ogni thread ha il proprio controllo del flusso, si è resa necessaria l'aggiunta di un secondo program counter. È stata duplicata anche la tabella che mappa i registri architetturali (`EAX`, `EBX`, ecc.) nei registri fisici, nonché il controllore di interrupt, visto che i thread possono essere interrotti in modo indipendente.

Poi c'è la **condivisione ripartita delle risorse** (*partitioned resource sharing*), secondo cui le risorse hardware sono divise rigidamente tra i thread. Per esempio, se la CPU ha una coda tra due stadi funzionali della pipeline, metà delle posizioni della coda potrebbero essere dedicate al thread 1, l'altra metà al thread 2. La ripartizione delle risorse è facile da ottenere, non richiede informazioni addizionali e fa sì che i thread non vadano in conflitto. Se tutte le risorse sono ripartite è come avere due CPU separate. La ripartizione ha anche un aspetto negativo: può succedere facilmente che un thread non usi alcune delle sue risorse che farebbero invece comodo a un altro thread, che però non ha diritto di usarle. Di conseguenza, alcune risorse che sarebbero potute servire a una qualche attività restano invece inoperose.

La **condivisione totale delle risorse** (*full resource sharing*) è il contrario della condivisione ripartita. Secondo questo schema, ogni thread può acquisire tutte le risorse di cui ha bisogno: il primo che arriva si serve da sé. Immaginiamo però due thread, uno veloce che fa solo somme e sottrazioni, uno più lento che svolge moltiplicazioni e divisioni. Se le istruzioni sono caricate dalla memoria con un ritmo maggiore rispetto alla velocità di esecuzione delle moltiplicazioni e delle divisioni, il numero d'istruzioni arretrate del thread lento (che

---

[1] Per elaborazione in *background* ("sullo sfondo") si intende l'elaborazione a bassa priorità di programmi che non hanno bisogno di interazioni con l'utente, e che quindi sono spesso a lui invisibili (*N.d.T.*).

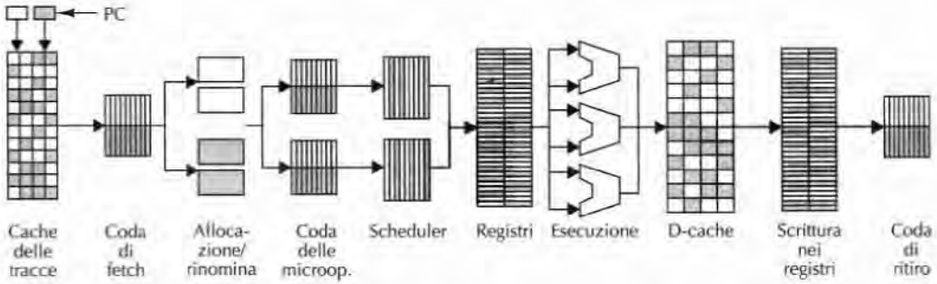


Figura 8.9 Condivisione delle risorse tra i thread della microarchitettura Netburst del Pentium 4.

si trova nella coda d'attesa per l'inserimento nella pipeline) crescerà di continuo. A un certo punto le istruzioni arretrate riempiranno completamente la coda d'attesa, costringendo il thread veloce ad arrestarsi per mancanza di spazio nella coda delle istruzioni. La condivisione totale delle risorse risolve il problema delle risorse inattive richieste da un thread, ma crea un problema nuovo in cui un thread può impossessarsi di così tante risorse da rallentare l'altro o addirittura da farlo fermare.

Uno schema intermedio è la **condivisione a soglia** (*threshold sharing*), secondo cui un thread può acquisire le risorse dinamicamente (non c'è ripartizione fissa), ma solo fino a un certo valore massimo. Questo approccio permette una certa flessibilità ed esclude il pericolo che un thread vada in *starvation*[2] a causa della sua incapacità di procurarsi le risorse. Per esempio, se nessun thread può acquisire più dei 3/4 della coda delle istruzioni, il thread più veloce potrà sempre proseguire la propria esecuzione, qualsiasi cosa faccia il thread più lento.

L'hyperthreading del Pentium 4 usa strategie di condivisione differenti a seconda delle risorse, nel tentativo di far fronte ai problemi cui si è accennato. La duplicazione è utilizzata per le risorse di cui i thread hanno sempre bisogno, come il program counter, la mappa dei registri e il controllore di interrupt. La duplicazione di queste risorse richiede un incremento della superficie del chip del 5%, un prezzo accettabile per il multithreading. Le risorse che sono così abbondanti per cui non si corre il rischio di acquisizione da parte di un solo thread, come le linee di cache, sono totalmente condivise in maniera dinamica. Invece le risorse che controllano il funzionamento della pipeline, come le varie code presenti al suo interno, sono ripartite esattamente tra i due thread. La Figura 8.9 illustra la pipeline principale della microarchitettura Netburst usata nel Pentium 4, laddove le caselle grigie e quelle bianche indicano l'allocazione delle risorse tra il thread grigio e quello bianco.

Nella figura notiamo che tutte le code sono ripartite a metà tra i thread, perciò nessuno dei due thread può soffocare l'altro. Anche i registri di allocazione e di rinomina sono ripartiti. Lo scheduler è condiviso dinamicamente con una soglia, per evitare che un thread reclami per sé l'intera coda. Gli altri stadi della pipeline sono totalmente condivisi.

---

[2] *To starve* vuol dire "morire di fame"; in questo contesto indica l'inattività involontaria di un thread dovuta all'esaurimento delle risorse (*N.d.T.*).

C'è da dire che il multithreading non è la panacea, ma che esiste anche un suo aspetto negativo. Se la ripartizione delle risorse è facile da implementare, la loro condivisione dinamica, specie se basata su soglia, richiede lavoro di amministrazione per monitorarne l'uso in fase d'esecuzione. Oltre a ciò, esistono delle situazioni in cui i programmi funzionano molto peggio con il multithreading che non senza. Per esempio, si immaginino due thread che, per funzionare bene, hanno bisogno entrambi di 3/4 della cache. Se eseguiti separatamente, girano senza problemi con pochi (costosi) fallimenti di cache. Se eseguiti insieme, ciascuno di loro provoca molti fallimenti di cache e il risultato complessivo potrebbe essere di gran lunga peggiore.

Si possono trovare altre informazioni sul multithreading del Pentium 4 in (Gerber e Binstock, 2004; Koufaty e Marr, 2003; Tuck e Tullsen, 2003).

8.1.3 Multiprocessori in un solo chip

Anche se il multithreading consente un incremento significativo delle prestazioni a un costo contenuto, per certi tipi di applicazioni si richiedono prestazioni molto superiori di quelle che può assicurare il multithreading. A tal fine è diretto lo sviluppo dei multiprocessori (chip con due o più CPU), che suscita particolare interesse nel mercato dei server di fascia alta e dell'elettronica di consumo. Diamo una rapida scorsa a entrambi i settori.

Multiprocessori omogenei in un solo chip

Grazie ai progressi della tecnologia VLSI è oggi possibile inserire in un chip due o più CPU potenti. Queste CPU si definiscono multiprocessori perché condividono le stesse cache, di primo e secondo livello, e la memoria principale (come illustrato nel Capitolo 2). Comunemente trovano applicazione nelle server farm che raggruppano molti server web. La capacità di far coabitare due CPU in un unico sistema, condividendo non solo la memoria, ma anche il disco e le interfacce di rete, permette spesso di raddoppiare le prestazioni del server senza raddoppiarne i costi (anche se il costo della CPU raddoppia, il suo prezzo è solo una frazione di quello dell'intero sistema).

Esistono due tipologie predominanti per il progetto di multiprocessori in un solo chip di piccole dimensioni. La prima è mostrata nella Figura 8.10(a): c'è davvero un solo chip, ma è dotato di una duplice pipeline che gli permette di raddoppiare potenzialmente il throughput. La Figura 8.10(b) mostra il secondo tipo di progetto: nel chip ci sono due *core*

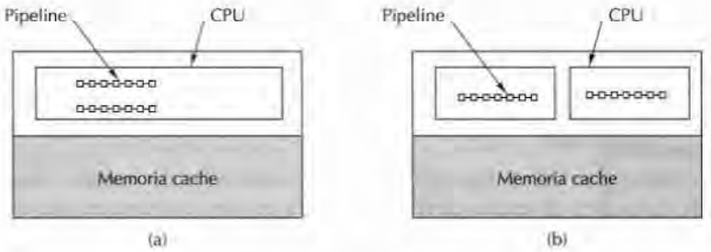


Figura 8.10 Multiprocessori a singolo chip. (a) Un chip con pipeline duplice. (b) Un chip con due core.

separati, ciascuno contenente un'intera CPU. Un **core** ("nucleo, cuore") è un grosso circuito, quale una CPU, un controllore di I/O o una cache, che può essere inserito su di un chip in maniera modulare, spesso uno di fianco all'altro.

Il primo progetto permette la condivisione di alcune risorse, come le unità funzionali, così che una CPU possa sfruttare le risorse inutilizzate da parte dell'altra CPU. Questo approccio però richiede la riprogettazione del chip e non si applica altrettanto bene al crescere del numero di CPU. Per contro, l'inserzione di due o più core CPU sullo stesso chip è relativamente facile.

L'argomento multiprocessori sarà trattato ancora in questo capitolo. Anche se l'analisi sarà focalizzata sui multiprocessori costruiti a partire da chip con CPU singola, si può estendere in larga misura anche ai chip con più CPU.

Multiprocessori eterogenei in un solo chip

Una gamma completamente diversa di multiprocessori in un solo chip è costituita dai sistemi integrati e in special modo dall'elettronica di consumo per contenuti audiovisivi, come apparecchiature TV, lettori DVD, videocamere, console per i videogiochi, telefoni cellulari e così via. Si tratta di sistemi che devono garantire prestazioni impegnative nel rispetto di forti vincoli. Anche se questi dispositivi sembrano a prima vista diversi, in realtà molti di loro non sono altro che piccoli calcolatori dotati di una o più CPU, di memorie, di controllori di I/O e di dispositivi di I/O specifici. Un cellulare, per esempio, contiene una CPU, una memoria, una piccola tastiera, un microfono, un altoparlante e una connessione senza fili alla rete, il tutto in un piccolo involucro.

Si consideri un lettore DVD portatile. Il computer al suo interno deve gestire le funzioni seguenti.

1. Controllo di un servomeccanismo economico e poco affidabile per il puntamento della testina.
2. Conversione da analogico a digitale.
3. Correzione degli errori.
4. Decodifica crittografica e gestione dei diritti digitali.
5. Decompressione video MPEG-2.
6. Decompressione audio.
7. Codifica dell'output per i sistemi televisivi NTSC, PAL o SECAM.

Tutte queste azioni vanno svolte rigorosamente in tempo reale, con vincoli circa la qualità del servizio, l'assorbimento di energia, la dissipazione del calore, le dimensioni, il peso e il prezzo del dispositivo.

I DVD contengono una lunga spirale d'informazioni (analogamente alla Figura 2.24 relativa ai CD). La testina di lettura deve puntare alla spirale con cura mentre il disco ruota. Si riesce a contenere il prezzo del dispositivo grazie alla progettazione di un meccanismo relativamente semplice e al controllo software della posizione della testina. Il segnale analogico proveniente dalla testina deve essere digitalizzato prima della sua elaborazione. Dopo la conversione è necessaria una pesante correzione degli errori perché i DVD ne contengono molti che devono essere corretti via software. La compressione video usa lo standard MPEG-2 che richiede un calcolo abbastanza complesso (simile alla trasformata di Fourier) per la decompressione. La compressione audio si avvale di un modello psicologico dell'udito e anch'esso richiede calcoli sofisticati in fase di decompressione. Infine, audio e video devono essere trasformati in un formato adatto al televisore e che varia da paese a paese: NTSC, PAL o SECAM. Non sorprende quindi come sia semplicemente impossibile svolgere tutte queste attività in tempo reale, via software e su CPU per uso generale. Ciò di cui c'è bisogno è un multiprocessore eterogeneo contenente più core, ciascuno specializzato per un certo compito. La Figura 8.11 presenta un esempio di lettore DVD.

Le funzioni svolte dai core della Figura 8.11 sono tutte diverse; ciascun core è stato progettato con cura per svolgere il proprio compito al meglio e costare il meno possibile. Per esempio, il video dei DVD è compresso con **MPEG-2** (da *Motion Picture Experts Group*, il gruppo di esperti che lo ha inventato). Si procede scomponendo ogni fotogramma (*frame*) in blocchi di pixel e applicando una complicata trasformazione matematica a ogni blocco. Un fotogramma può essere fatto interamente di blocchi trasformati, oppure si può specificare che un determinato blocco sia uguale a un blocco del fotogramma precedente, localizzato a un offset $(\Delta x, \Delta y)$ rispetto alla sua posizione attuale, fatta eccezione per una manciata di pixel cambiati. Un calcolo di questo tipo sarebbe molto lento via software, ma è possibile costruire dei circuiti di decodifica MPEG-2 che possono svolgerlo in modo abbastanza rapido. Analogamente, anche la decodifica audio e la ricodifica composita audio-video del segnale (per rispettare gli standard televisivi mondiali) possono essere svolte con maggior successo da processori dedicati. Queste considerazioni giustificano largamente la progettazione di chip multiprocessori contenenti core specializzati per applicazioni audiovisive. Il processore di controllo è una CPU programmabile di uso generale, perciò il chip multiprocessore può essere usato anche per altre applicazioni simili.

Il telefono cellulare è un altro dispositivo che necessita di un multiprocessore eterogeneo per il suo funzionamento. I modelli attuali possono includere macchine fotografiche, videocamere, giochi, navigatori web, lettori di posta, ricevitori radio di segnali satellitari digitali; essi si basano sulla tecnologia cellulare (CDMA o GSM) o su Internet senza fili (IEEE 802.11, detta anche tecnologia WiFi). I modelli futuri saranno dotati di tutti questi dispo-

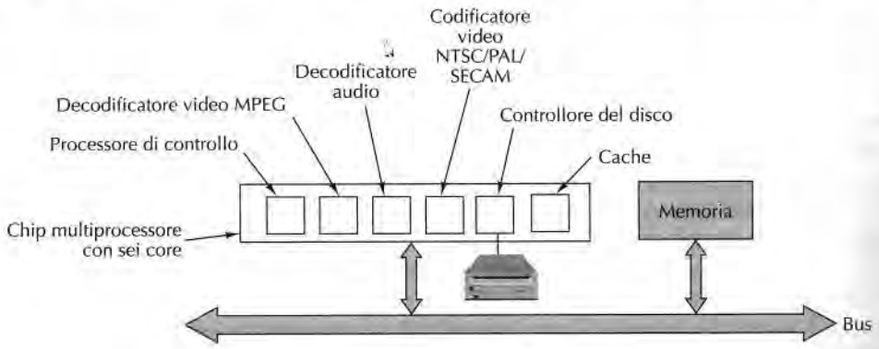


Figura 8.11 Struttura logica dei lettori DVD contenente un multiprocessore eterogeneo con vari core dedicati.

sitivi. Gli apparecchi acquistano sempre più funzionalità, gli orologi diventano mappe basate su GPS, gli occhiali diventano radio e la richiesta di multiprocessori eterogenei continuerà a crescere.

Verrà presto il giorno in cui i chip avranno 500 milioni di transistor. Si tratterebbe di una dimensione troppo grande perché si possa progettare a partire dai circuiti elementari. Lo sforzo umano richiesto sarebbe tale da rendere un chip obsoleto ancor prima di essere ultimato. L'unico modo di procedere è usare i core (di fatto delle librerie), ciascuno contenente un sottoassemblato abbastanza grande, e collegarli tramite il montaggio su un chip. Ai progettisti resta solo da decidere quale core CPU usare per il processore di controllo e quali processori specializzati associargli per assisterlo. Spostare il carico di lavoro sul software che gira sul processore di controllo rallenta il sistema, ma consente chip più piccoli (e più economici). La presenza di processori diversi per l'elaborazione audio e video aumenta la superficie del chip e fa lievitare i costi, ma produce prestazioni migliori a parità di frequenza di clock, il che implica un minor consumo energetico e una minore dissipazione del calore. Di conseguenza, i progettisti si trovano sempre più spesso di fronte a questo tipo di compromessi macroscopici, piuttosto che alle prese con i dettagli realizzativi dei componenti hardware.

Le applicazioni audiovisive lavorano su grosse quantità di dati che devono essere elaborate velocemente, perciò la memoria (sotto varie forme) occupa dal 50% al 75% della superficie dei chip e questa percentuale è destinata a salire. Si pongono molti interrogativi di progetto: quanti livelli di cache prevedere? Usare una cache specializzata o una unificata? Quanto dovrebbe essere grande ciascuna cache? Quanto veloce? Vale la pena inserire parte della memoria nel chip? Meglio memoria SRAM o SDRAM? Le risposte a queste domande hanno forti implicazioni sulle prestazioni, sul consumo energetico e sulla dissipazione del calore del chip.

Al di là del progetto dei processori e dei sistemi di memoria, un altro tassello importante per le sue conseguenze è il sistema di comunicazione: come far comunicare tutti i core tra di loro? Nel caso di piccoli sistemi basterà un solo bus, ma nei sistemi grandi diverrebbe subito un collo di bottiglia. In genere il problema può essere risolto optando per più bus o per un anello che attraversa tutti i core. Nell'ultimo caso, l'arbitraggio avviene tramite il passaggio di un piccolo pacchetto chiamato **token** (gettone) lungo l'anello. Per poter trasmettere, un core deve per prima cosa catturare il token; quando ha finito di comunicare lo può reintrodurre sull'anello perché riprenda a circolare. Questo protocollo previene le collisioni sull'anello.

La Figura 8.12 presenta **CoreConnect** di IBM, un esempio di interconnessione nel chip. È un'architettura per la connessione di core su multiprocessori eterogenei su un chip, usata soprattutto per la progettazione di chip contenenti interi sistemi. In un certo senso, CoreConnect sta ai multiprocessori nel singolo chip come il bus PCI sta al Pentium, ovvero è la colla che tiene uniti i vari pezzi. Però, a differenza del bus PCI, CoreConnect è stato progettato senza vincoli di retrocompatibilità dovuti a componenti o protocolli precedenti, e senza i vincoli propri dei bus che risiedono sulle schede, come il limite sul numero di piedini presenti sul connettore.

CoreConnect è composto da tre bus. Quello del **processore** è un bus veloce, sincrono e a pipeline con linee dati di 32, 64 o 128 bit, temporizzate a 66, 133, o 183 MHz. La velocità massima di trasferimento è 23,4 Gbps (contro 4,2 Gbps per il bus PCI). Il funzionamento a pipeline permette ai core di richiedere il bus mentre è già impegnato in un trasferimento e consente a core diversi di usare contemporaneamente linee dati distinte, come per il bus PCI. Il bus del processore è ottimizzato per il trasferimento di blocchi corti ed è concepito per connettere core veloci, come la CPU, il decodificatore MPEG-2, le reti ad alta velocità o componenti similari.

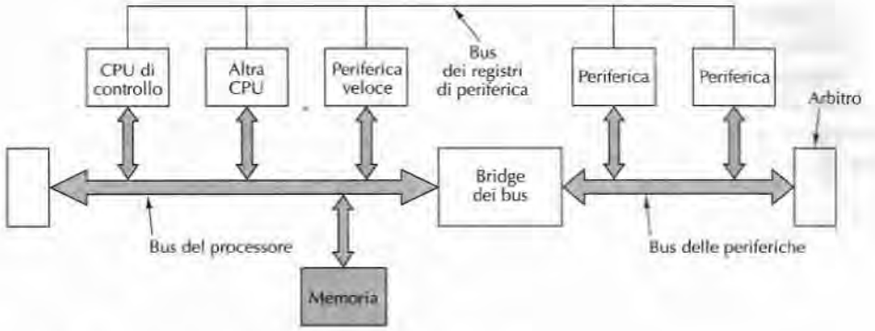


Figura 8.12 Esempio di architettura CoreConnect di IBM.

Se il bus del processore fosse esteso a tutto il chip le sue prestazioni degraderebbero, perciò esiste un secondo bus per i dispositivi di I/O più lenti, come gli UART, i timer, i controllori USB e le porte seriali. Questo bus delle periferiche è stato progettato per avere un'interfaccia semplice con le periferiche a 8, 16 o 32 bit e per contenere non più di qualche centinaio di porte logiche. Anch'esso è sincrono, ma la velocità massima di trasferimento è di 300 Mbps. I due bus sono collegati attraverso un *bridge* simile a quello usato qualche anno fa per collegare i bus PCI e ISA nei PC, prima che gli ISA divenissero obsoleti.

Il terzo bus dei **registri di periferica** è molto lento, asincrono e usa un *handshaking* per permettere a ogni processore di accedere ai registri di tutte le periferiche per il controllo dei rispettivi dispositivi. È concepito per trasferimenti poco frequenti e di pochi byte alla volta.

IBM CoreConnect ha definito uno standard per i bus nel chip, per la loro interfaccia e per l'intelaiatura generale del chip con l'ambizione di creare una versione in miniatura del mondo PCI, in cui ogni produttore possa costruire processori e controllori che si interconnettano con facilità. Esiste però una differenza: nel mondo PCI i produttori costruiscono e vendono schede acquistabili dai distributori e dagli utenti finali. Nel mondo CoreConnect i core sono progettati da terze parti che però non realizzano i prodotti, ma li brevettano come proprietà intellettuale e vendono le licenze alle aziende di elettronica di consumo, che progettano i chip multiprocessori eterogenei basandosi sui core propri e su quelli sottoposti a licenza di terzi. Poiché la produzione di chip così grandi e complessi richiede investimenti massicci in impianti produttivi, nella maggior parte dei casi le aziende di elettronica di consumo si limitano alla progettazione e affidano la realizzazione del chip a produttori di semiconduttori. Esistono numerosi core di CPU (ARM, MIPS, PowerPC, e così via), di decodificatori MPEG, di processori di segnali digitali e per tutti i controllori di I/O standard.

Oltre a CoreConnect IBM, anche **AMBA** (*Advanced Microcontroller Bus Architecture*) è usato largamente (Flynn, 1997). Altri bus di questo tipo, però meno diffusi, sono **VCI** (*Virtual Component Interconnect*) e **OCP-IP** (*Open Core Protocol-International Partnership*), in competizione tra loro per una quota di mercato (Kogel e Meyr, 2004; Ouadjaout e

Houzet, 2004). E questo è soltanto l'inizio: c'è già chi pensa alle reti nel chip (Benini e De Micheli, 2002).

A causa delle difficoltà legate ai problemi di dissipazione del calore (dovuti a loro volta a velocità di clock sempre maggiori) i multiprocessori nel chip sono oggi un tema attuale; rimandiamo a (Claasen, 2003; Jerraya e Wolf, 2005; Kumar et al., 2004; Lavagno, 2002; Lines, 2004; Ravikumar, 2004) per ulteriori informazioni.

8.2 Coprocessori

Dopo aver esaminato i modi del parallelismo nel chip, facciamo un passo in avanti e ci interessiamo al modo di velocizzare un calcolatore mediante l'aggiunta di un secondo processore specializzato. Esistono molte varianti di **coprocessori**, di dimensioni molto diverse. Sui mainframe IBM 360, e sui loro successori, esistono canali di I/O indipendenti per l'input e l'output. Allo stesso modo, il CDC 6600 disponeva di 10 processori indipendenti per le operazioni di I/O. Anche la grafica e l'aritmetica in virgola mobile si prestano all'uso di coprocessori, e lo stesso DMA può essere visto come un coprocessore. In alcuni casi la CPU assegna al coprocessore un'istruzione o un insieme d'istruzioni e gli ordina di eseguirle; in altri casi il coprocessore è più indipendente e lavora per conto proprio.

Dal punto di vista fisico, i coprocessori possono costituire apparati separati (i canali di I/O del 360), trovarsi su schede a innesto (i processori di rete) o occupare parte del chip principale (coprocessore in virgola mobile). In tutti i casi, ciò che li caratterizza è il fatto di assistere nell'esecuzione un altro processore, che resta il processore principale. Esaminiamo ora tre settori in cui è possibile velocizzare le prestazioni: elaborazione di rete, multimedia e crittografia.

8.2.1 Processori di rete

Oggigiorno molti calcolatori sono connessi a una rete o a Internet. A seguito del progresso tecnologico dell'hardware di rete, le reti sono oggi così veloci che è sempre più difficile elaborare via software i dati in ingresso e in uscita. A questo riguardo sono stati sviluppati speciali processori di rete per gestire il traffico; molti calcolatori di fascia alta sono dotati di processori di questo tipo. In questo paragrafo cominciamo con l'introdurre le reti, per poi illustrare il funzionamento dei processori di rete.

Introduzione alle reti

Le reti di calcolatori sono di due tipi: **LAN** (*Local-Area Network*, "rete locale"), che connettono più computer all'interno di un edificio o di un campus universitario, e **WAN** (*Wide-Area Network*, "rete geografica"), per la connessione di calcolatori che si trovano a grande distanza. **Ethernet** è la LAN più diffusa: inizialmente era costituita da cavi spessi contenenti un filo per ogni computer, collegato tramite una spina che veniva chiamata eufemisticamente **spina a vampiro** (*vampire tap*). L'odierna Ethernet prevede che i calcolatori siano collegati a un commutatore (*switch*) centrale, come schematizzato nella Figura 8.13. In origine, Ethernet *si trascinava* a 3 Mbps, ma la prima versione commerciale raggiungeva già i 10 Mbps. È stata poi sostituita da Ethernet veloce (*fast Ethernet*) a 100 Mbps e successivamente da

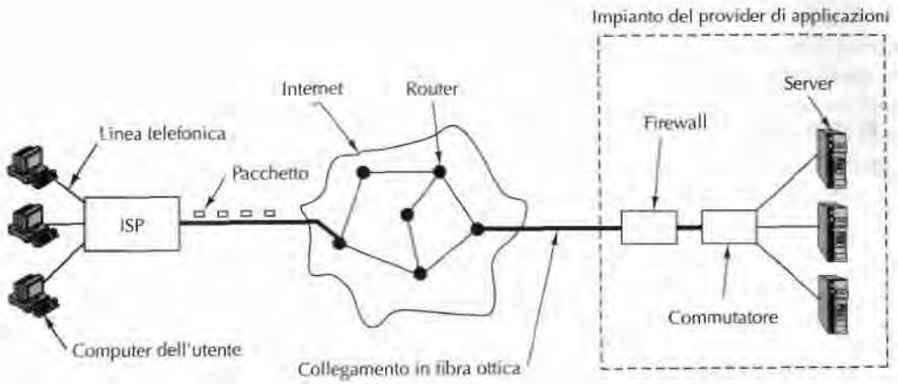


Figura 8.13 Connessione degli utenti ai server di Internet.

gigabit Ethernet a 1 Gbps. Esiste già sul mercato una versione di Ethernet a 10 Gbps e la prossima sarà a 40 Gbps.

Le WAN sono organizzate in modo diverso. Sono costituite da computer per l'instradamento, detti **router**, collegati tramite cavi o fibre ottiche, come illustrato al centro della Figura 8.13. I dati sono suddivisi in frammenti chiamati **pacchetti** (*packet*) che vanno in genere dai 64 ai 1500 byte e che vengono trasferiti dalla macchina mittente alla destinataria passando attraverso uno o più router. A ogni passaggio (*hop*, "salto"), il pacchetto è memorizzato in un router e quindi inoltrato al successivo lungo il cammino non appena la linea di trasmissione lo consenta. Questa tecnica si chiama **commutazione di pacchetto store-and-forward**.

Anche se molte persone pensano a Internet nei termini di un'unica WAN, dal punto di vista tecnico è invece un insieme di molte WAN interconnesse. In ogni caso, questa distinzione non è rilevante ai nostri scopi. La Figura 8.13 fornisce una visione panoramica di Internet dal punto di vista dell'utente domestico. Il suo calcolatore è collegato a un server web tramite una linea telefonica, per mezzo di modem a 56 Kbps o ADSL, come illustrato nel Capitolo 2 (in alternativa è possibile una connessione che usa la linea della TV via cavo, nel qual caso la parte sinistra della figura cambia leggermente e il *provider* diventa la compagnia che fornisce la TV via cavo). Il calcolatore dell'utente spezzetta i dati da inviare al server in pacchetti e li spedisce all'**ISP** (*Internet Service Provider*, "fornitore di servizi Internet") dell'utente, una società che garantisce ai propri utenti l'accesso a Internet. L'ISP ha in genere una connessione ad alta velocità (di solito una fibra ottica) verso una delle reti regionali o una *backbone* (dorsale) di Internet. I pacchetti dell'utente sono inoltrati nella rete un salto alla volta finché non raggiungono il server web.

Molte società che forniscono servizi web dispongono di un **firewall** ("muro di fuoco"), un computer specializzato che filtra tutto il traffico in ingresso e cerca di rimuovere i pacchetti indesiderati (per esempio quelli che provengono da hacker che cercano di introdursi nel server). Il firewall è collegato alla LAN, in genere a un commutatore Ethernet, che instrada i pacchetti al server desiderato. Naturalmente, le cose nella realtà sono molto più complesse di come le abbiamo presentate, ma l'idea sostanziale della Figura 8.13 resta valida.

Il software di rete è organizzato per **protocolli**, ciascuno dei quali comprende un insieme di formati, di sequenze di scambio e di regole sul significato dei pacchetti. Per esempio, quando un utente vuole scaricare una pagina web da un server, il suo applicativo di navigazione (*browser*) spedisce al server un pacchetto contenente la richiesta *GET PAGE* e che usa **HTTP** (*HyperText Transfer Protocol*, "protocollo di trasferimento di ipertesti"); il server recepisce la richiesta e la elabora. Esistono molti protocolli e spesso vengono usati in combinazione. Nella maggior parte delle situazioni i protocolli sono strutturati come una serie di livelli (*layer*), in cui i pacchetti vengono passati dai livelli superiori a quelli inferiori perché li elaborino e, una volta raggiunto il livello più basso, perché siano trasmessi. Dal lato destinatario, i pacchetti si fanno strada tra i livelli in ordine inverso, dal basso verso l'alto.

Dal momento che l'elaborazione dei protocolli è l'attività principale dei processori di rete, dobbiamo dilungarci ancora un po' sull'argomento prima di passare ai processori veri e propri. Torniamo per un momento alla richiesta *GET PAGE*. Come avviene la sua spedizione al server web? In pratica il software di navigazione per prima cosa stabilisce una connessione con il server web basata sul protocollo **TCP** (*Transmission Control Protocol*, "protocollo di controllo della trasmissione"). Il software che implementa questo protocollo verifica che i pacchetti siano stati ricevuti tutti correttamente e nell'ordine giusto. Se un pacchetto viene perso, TCP garantisce che verrà ritrasmesso finché non raggiunga il destinatario.

Il navigatore web compila la richiesta *GET PAGE* come un messaggio HTTP e lo passa al software TCP per la trasmissione in rete. Il software TCP aggiunge all'inizio del messaggio un'intestazione contenente una serie di numeri e altre informazioni, che ovviamente prende il nome di **intestazione TCP** (*TCP header*).

Quando ha finito la sua attività, il software TCP passa l'intestazione e la parte di messaggio vero e proprio (il *payload*, che in questo caso contiene la richiesta *GET PAGE*) a un altro componente software che implementa il **protocollo IP** (*Internet Protocol*). Questo software antepone al messaggio un'**intestazione IP** contenente l'indirizzo del mittente (la macchina da cui proviene il pacchetto) e quello del destinatario (la macchina cui il pacchetto è diretto), il massimo numero di passaggi cui può sopravvivere (per evitare che i pacchetti sopravvivano indefinitamente), una checksum (per rilevare errori di trasmissione o di memoria), e altri campi.

Il pacchetto risultante (dall'intestazione IP, intestazione TCP e richiesta *GET PAGE*) viene passato in basso a livello di trasmissione dei dati (*data link layer*), in cui gli viene allegata un'ulteriore intestazione che serve alla trasmissione vera e propria. Il livello di trasmissione dei dati allega in fondo al messaggio anche una checksum che si chiama **CRC** (*Cyclic Redundancy Check*, "controllo a ridondanza ciclica") e che serve per rilevare errori di trasmissione. Potrebbe sembrare ridondante la presenza di due checksum a livello di trasmissione dei dati e a livello IP, ma serve a migliorare l'affidabilità. A ogni salto lungo la rete viene controllato il valore di CRC, quindi viene rimosso e rigenerato (insieme all'intestazione) secondo un formato adatto al collegamento in uscita. La Figura 8.14 mostra le sembianze del pacchetto quando si trova in una Ethernet; il pacchetto in una linea telefonica è simile, eccetto per il fatto che viene preceduto da una "intestazione di linea telefonica" invece che da un'intestazione Ethernet. La gestione delle intestazioni è importante ed è una delle mansioni dei processori di rete. È evidente che abbiamo appena sfiorato la superficie di una materia sconfinata quali sono le reti di computer. Per una trattazione più dettagliata si faccia riferimento a (Tanenbaum, 2003).

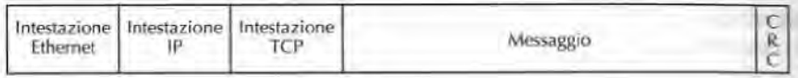


Figura 8.14 Schema dei pacchetti Ethernet.

Introduzione ai processori di rete

Le reti collegano molti tipi di dispositivi. Gli utenti domestici si collegano tramite PC (desktop o portatili), ma anche (e sempre più) tramite console di videogiochi, PDA (computer palmari) e telefoni cellulari; le aziende sono collegate tramite PC o server. D'altra parte, esistono anche molti dispositivi che svolgono nella rete una funzione d'intermediazione, tra cui router, commutatori, firewall, proxy web e bilanciatori di carico. È interessante notare che questi sistemi intermediari richiedono una quantità di risorse preponderante nella rete, dal momento che trasferiscono il maggior numero di pacchetti al secondo. Anche i server sono abbastanza esigenti, a differenza dei calcolatori degli utenti domestici.

Un pacchetto ricevuto può richiedere vari tipi di elaborazione, a seconda della rete o del pacchetto stesso, prima di essere inoltrato lungo la linea in uscita o prima di essere consegnato a un programma applicativo. Alcune elaborazioni possibili sono: decidere dove spedire il pacchetto, scomporlo o riassemblarne i pezzi, gestire la sua qualità di servizio (specialmente per i flussi audio o video) o la sua sicurezza (per esempio cifrandolo o decifrandolo), comprimerlo o espanderlo, e così via.

In una LAN a 40 Gbps e con pacchetti di 1 KB, un computer di rete potrebbe dover elaborare quasi 5 milioni di pacchetti al secondo. Con pacchetti di 64 byte si arriva a circa 80 milioni di pacchetti al secondo. Lo svolgimento delle attività di cui sopra nel giro di 12-200 ns è semplicemente impossibile per il software (per non parlare del fatto che di ogni pacchetto servono più copie): è fondamentale l'assistenza dell'hardware.

Una possibile soluzione per elaborare velocemente i pacchetti prevede l'utilizzo di un **ASIC** (*Application-Specific Integrated Circuit*, "circuito integrato per applicazione specifica") progettato su misura. Un chip siffatto è una specie di programma cablato in grado di calcolare l'insieme di funzioni per cui è stato progettato. Attualmente molti router usano circuiti ASIC. Tuttavia i chip ASIC hanno numerosi problemi: innanzitutto richiedono molto tempo sia in fase di progettazione, sia in fase di produzione. Inoltre sono rigidi, perciò se gli si vuole aggiungere una funzionalità bisogna progettare e costruire un chip ex novo. La gestione dei bachi poi è un incubo, visto che l'unico modo per riparare un ASIC è progettare, costruire, spedire e installare un chip nuovo. Infine, si tratta di chip molto costosi, a meno che se ne produca un quantitativo così grande da ammortizzare i costi di sviluppo.

Un'altra possibile soluzione usa i circuiti **FPGA** (*Field Programmable Gate Array*, "circuito a matrice programmabile"), costituiti da un certo numero di porte che possono essere organizzate in un circuito mediante la modifica dei collegamenti tra di loro. Lo sviluppo di questi chip impiega molto meno tempo rispetto agli ASIC e in più gli FPGA possono essere ricablati sul campo: basta rimuoverli dal sistema e inserirli in un dispositivo speciale per la loro riprogrammazione. D'altra parte sono circuiti complessi, lenti e costosi, il che fa di loro circuiti poco richiesti, se non per certe applicazioni di nicchia.

Infine veniamo ai **processori di rete**, dispositivi programmabili che possono gestire i pacchetti in ingresso e in uscita alla stessa velocità con cui viaggiano sui collegamenti (cioè

in tempo reale). Un progetto tipico vede il processore di rete posto su una scheda a innesto, insieme a una memoria e ai circuiti logici di supporto. La scheda è collegata a una o più linee di rete che sono dirette al processore di rete, che estrae i pacchetti, li elabora e li spedisce su di una linea diversa (se è un router) o lungo il bus principale di sistema (per esempio lungo il bus PCI) nel caso si tratti di un dispositivo che si trova sul PC di un utente finale. La Figura 8.15 illustra la scheda e il chip di un comune processore di rete.

La scheda è provvista di memorie SRAM e SDRAM, usate in genere in modi diversi. La SRAM è più veloce e più costosa della SDRAM, perciò la sua disponibilità è molto limitata. La SRAM si usa per contenere le tabelle d'instradamento (*routing table*) e altre strutture dati importanti, mentre la SDRAM memorizza i pacchetti in elaborazione. Grazie alla separazione tra il chip e le memorie, i progettisti possono decidere liberamente le quantità di SRAM e di SDRAM da includere nella scheda. Così facendo, è possibile dotare di poca memoria le schede di fascia bassa con una sola linea in ingresso (destinate per esempio a un PC o a un server), mentre le schede di fascia alta, progettate per i grandi router, possono essere provviste di molta più memoria.

I chip dei processori di rete sono ottimizzati per elaborare velocemente un gran numero di pacchetti in ingresso e in uscita; in particolare, un router con una mezza dozzina di linee potrebbe trovarsi a elaborare milioni di pacchetti al secondo su ogni linea. L'unico modo per raggiungere queste velocità è di costruire processori di rete con molto parallelismo; infatti, ogni processore di rete contiene svariati **PPE** (acronimo di nomi diversi: *Protocol/Programmable/Packet Processing Engine*, "motore di elaborazione del protocollo/programmabile/di pacchetto"). Ogni PPE è un core RISC (eventualmente modificato) con un piccolo quantitativo di memoria per il programma e le variabili.

I motori PPE possono essere organizzati in due modi diversi. Il caso più semplice consiste nell'impiego di PPE identici: quando il processore di rete riceve un pacchetto (dalla rete o dal bus) lo smista a un PPE inattivo perché lo elabori. Se tutti i PPE sono impegnati, il pacchetto entra in una coda contenuta nella SDRAM della scheda, in attesa che si liberi un

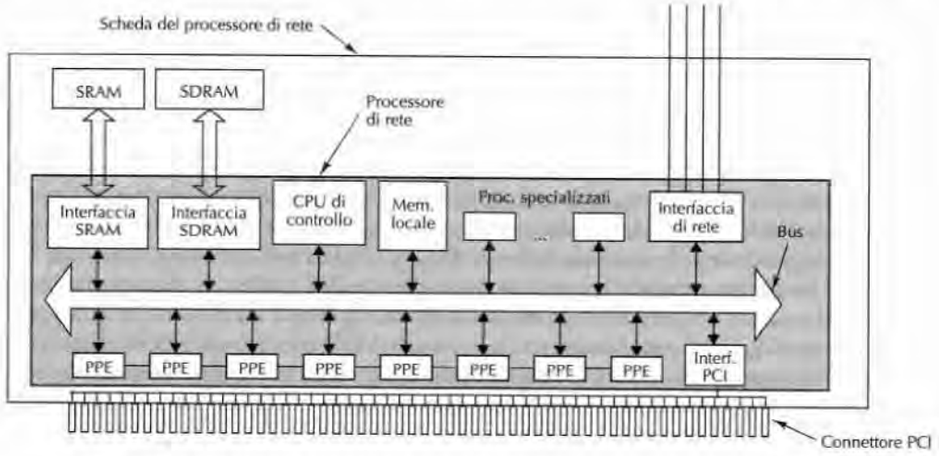


Figura 8.15 Scheda e chip di un comune processore di rete.

PPE. In un'organizzazione di questo tipo non ci sono i collegamenti orizzontali mostrati nella Figura 8.15 tra un PPE e l'altro, perché questi non hanno alcun bisogno di comunicare.

L'altro tipo di organizzazione dei PPE è a pipeline, e ogni motore esegue un passo di elaborazione e inserisce nel pacchetto in uscita un puntatore al PPE successivo nella pipeline. In questo modo, la pipeline dei PPE somiglia molto alle pipeline delle CPU che abbiamo studiato nel Capitolo 2. In entrambe le organizzazioni, i motori PPE sono interamente programmabili.

I progetti più avanzati prevedono PPE con multithreading, dotati perciò di diversi insiemi di registri e di un particolare registro che determina l'insieme correntemente in uso. Grazie a questa caratteristica è possibile eseguire più programmi alla volta e la commutazione da un programma (cioè da un thread) all'altro avviene semplicemente modificando la variabile "insieme dei registri corrente". Comunemente, un PPE commuta immediatamente a favore di un thread eseguibile non appena quello in esecuzione va in stallo, per esempio a seguito di un accesso alla SDRAM (che richiede diversi cicli di clock). Grazie a questo accorgimento è possibile utilizzare appieno i PPE anche quando si bloccano per frequenti accessi alla SDRAM o quando eseguono un'altra operazione esterna piuttosto lenta.

Oltre ai PPE, i processori di rete contengono sempre un processore di controllo (in genere una CPU RISC) impiegato per svolgere tutto quel lavoro che non ha a che fare con l'elaborazione dei pacchetti, come l'aggiornamento delle tabelle d'instradamento. Il suo programma e i suoi dati si trovano in memoria locale del chip. Inoltre, molti chip di rete contengono uno o più processori specializzati per l'esecuzione di operazioni di corrispondenza tra forme (*pattern matching*) o di altre operazioni critiche. Si tratta per lo più di piccoli ASIC in grado di effettuare una sola, semplice operazione, come la ricerca di un indirizzo di destinazione all'interno della tabella d'instradamento. Tutti i componenti del processore di rete comunicano per mezzo di uno o più bus paralleli del chip, con velocità di svariati Gigabit al secondo.

Elaborazione dei pacchetti

Alla sua ricezione, un pacchetto passa attraverso un certo numero di fasi di elaborazione, indipendentemente dal fatto che il processore di rete sia o meno organizzato a pipeline. Alcuni processori di rete dividono queste fasi in due tipi di operazioni, quelle da effettuare sui pacchetti in ingresso (dalla linea di rete o dal bus di sistema), dette perciò **elaborazione in entrata dei pacchetti di rete**, e quelle da effettuare sui pacchetti in uscita, che costituiscono l'**elaborazione in uscita dei pacchetti di rete**. Quando si utilizza questa distinzione, un pacchetto subisce prima l'elaborazione in entrata, poi quella in uscita. La demarcazione tra operazioni in entrata e in uscita è flessibile, perché alcune fasi possono essere svolte indifferentemente in uno dei due momenti (per esempio la raccolta delle statistiche sul traffico).

In seguito analizziamo un possibile ordinamento delle diverse fasi, ma si tenga presente che non tutti i pacchetti necessitano di tutte le fasi e che sono possibili altri ordinamenti ugualmente validi.

1. **Verifica della checksum.** Se il pacchetto in entrata proviene da Ethernet, viene ricalcolato il valore CRC e confrontato con quello nel pacchetto per verificare che non ci siano stati errori di trasmissione. Se il CRC di Ethernet è corretto o assente, viene ricalcolata la checksum IP e confrontata con quella del pacchetto IP per assicurarsi che

non sia stato danneggiato da un bit difettoso della memoria del mittente e che avrebbe lasciato traccia nella checksum ivi calcolata. Se le checksum sono corrette, il pacchetto viene accettato per essere ulteriormente elaborato; in caso contrario, viene semplicemente scartato.

2. **Estrazione di campi.** Viene analizzata la parte significativa dell'intestazione e vengono estratti i campi più importanti. All'interno di un commutatore Ethernet viene esaminata solo l'intestazione Ethernet, mentre in un router IP viene ispezionata l'intestazione IP. I campi chiave sono memorizzati nei registri (organizzazione con PPE paralleli) o nella SRAM (organizzazione a pipeline).
3. **Classificazione dei pacchetti.** Il pacchetto è classificato in base a una serie di regole programmabili. La classificazione più semplice distingue tra pacchetti di dati e di controllo, ma di solito si effettuano distinzioni molto più accurate.
4. **Selezione del percorso.** Molti processori di rete predispongono un percorso veloce e ottimizzato per la gestione dei pacchetti più comuni, mentre gli altri pacchetti sono trattati diversamente, spesso a opera del processore di controllo. Si effettua quindi una selezione tra il percorso veloce e quello lento.
5. **Determinazione del destinatario di rete.** I pacchetti IP contengono l'indirizzo del destinatario di 32 bit. Non è possibile, né tanto meno auspicabile, effettuare la ricerca della destinazione di ogni pacchetto IP all'interno di una tabella di 2^{32} elementi, perciò la parte più significativa dell'indirizzo IP viene usata per identificare il numero di rete, il resto per specificare una macchina in quella rete. I numeri di rete possono avere una lunghezza arbitraria, perciò la determinazione del numero della rete destinataria non è banale ed è complicata dal fatto che ci sono più corrispondenze possibili e che quella giusta è la più lunga. In genere questa fase è affidata a un circuito ASIC ad hoc.
6. **Instradamento.** Una volta determinata la rete del destinatario, si sceglie la linea d'uscita lungo cui instradare il pacchetto cercandola in una tabella della SRAM. Anche per questa fase si può usare un circuito ASIC.
7. **Scomposizione e riassemblaggio.** I programmi cercano di passare a livello TCP ingenti quantitativi di dati per ridurre il numero di chiamate di sistema, ma TCP, IP e Ethernet definiscono tutti una dimensione massima per i pacchetti che possono gestire. In conseguenza di queste limitazioni, i dati utili e i pacchetti vengono scomposti dal mittente e i pezzi risultanti vengono riassemblati dal destinatario. È uno dei compiti che possono essere svolti dal processore di rete.
8. **Elaborazione.** Alle volte sono necessari calcoli molto pesanti sui dati che costituiscono il messaggio (come compressione/decompressione o cifratura/decifratura). È un altro compito affidabile al processore di rete.
9. **Gestione dell'intestazione.** Può essere necessario aggiungere, rimuovere o modificare alcuni campi dell'intestazione. Per esempio, l'intestazione IP ha un campo che conta il numero di salti che il pacchetto può effettuare prima di venire scartato. Questo campo va decrementato dopo ogni ritrasmissione: un altro compito adatto al processore di rete.

10. **Gestione della coda.** I pacchetti in entrata o in uscita vengono spesso messi in coda mentre attendono il loro turno di elaborazione. Le applicazioni multimediali potrebbero aver bisogno di un certo intervallo tra un pacchetto e l'altro per evitare il fenomeno del *bitter*. Un firewall o un router potrebbero distribuire il carico di pacchetti in entrata su diverse linee in uscita in base ad alcune regole. Tutte queste attività possono essere svolte dal processore di rete.
11. **Generazione della checksum.** I pacchetti in uscita devono essere provvisti di checksum. La checksum IP può essere generata dal processore di rete, ma il valore di CRC Ethernet di solito viene calcolato dall'hardware.
12. **Contabilità.** In alcuni casi si richiede un po' di contabilità sul traffico di pacchetti, soprattutto quando una rete instrada traffico per conto di altre reti come servizio commerciale. Il processore di rete può occuparsi della contabilità.
13. **Raccolta di statistiche.** Infine, molte organizzazioni intendono accumulare statistiche circa il proprio traffico e il processore di rete si presta bene alla loro raccolta.

Incremento delle prestazioni

Le prestazioni sono fondamentali per i processori di rete: che cosa si può fare per migliorarle? Prima di rispondere, dobbiamo definire che cosa voglia dire miglioramento. Una metrica possibile è il numero di pacchetti inoltrati ogni secondo, un'altra è il numero di byte inoltrati al secondo. Sono due misure differenti e non è detto che uno schema che funziona bene con pacchetti piccoli funzioni altrettanto bene con pacchetti più grandi. In particolare, nel primo caso potrebbe giovare una riduzione del numero di destinazioni ricercate in ogni secondo, mentre ciò potrebbe non essere vero nel caso di pacchetti grandi.

Il modo più diretto per migliorare le prestazioni è aumentare la velocità di clock del processore di rete. Ovviamente il miglioramento non è lineare, perché può essere influenzato dal tempo di ciclo della memoria e da altri fattori. E poi, un clock più veloce vuol dire più calore da dissipare.

Una soluzione che molto spesso ripaga è l'aggiunta di più PPE e l'incremento del parallelismo, soprattutto quando si usa l'organizzazione parallela dei PPE. Anche l'allungamento della pipeline può aiutare, ma solo se l'intera elaborazione di un pacchetto sia suddivisibile in fasi più corte.

Un'altra tecnica possibile prevede l'adozione di un processore specializzato o di un ASIC per gestire specificatamente le operazioni che richiedono molto tempo, che vengono eseguite ripetutamente e la cui l'esecuzione via hardware risulta molto più veloce di quella via software. Tra i candidati ci sono sicuramente le ricerche nelle tabelle, il calcolo della checksum e la cifratura.

L'aggiunta di un maggior numero di bus interni e l'ampliamento dei bus esistenti può aiutare ad aumentare la velocità contribuendo ad accelerare lo spostamento dei pacchetti nel sistema. Infine, tra i miglioramenti si può considerare anche la sostituzione della SDRAM con memoria SRAM, ma che comporta costi aggiuntivi.

Naturalmente resta molto da dire sui processori di rete. Alcuni riferimenti in tal senso sono (Comer, 2005; Crowley et al., 2002; Lekkas, 2003; Papaefstathiou et al., 2004).

8.2.2 Processori multimediali

Un altro settore in cui si rendono utili i coprocessori è quello della gestione di immagini fotografiche e di flussi audio o video ad alta risoluzione. Le normali CPU non sono molto dotate per elaborare volumi di dati così ingenti come quelli richiesti da queste applicazioni. Perciò alcuni PC odierni dispongono di coprocessori multimediali cui affidare una larga parte del lavoro, e ciò diverrà la norma nei PC futuri.

Il processore multimediale Nexperia

Ci occupiamo di questo settore, la cui importanza è in continua ascesa, attraverso un esempio concreto: i Philips Nexperia, una famiglia di chip disponibili a diverse frequenze di clock. **Nexperia** è un multiprocessore eterogeneo in un chip autocontenuto nel senso della Figura 8.11. Contiene diversi core, tra cui una CPU VLIW TriMedia per il controllo, ma anche numerosi core per l'elaborazione audio, video, di immagini o di rete. Può essere usato come processore autonomo principale all'interno di lettori o registratori CD, DVD o MP3, negli apparecchi TV, nelle macchine fotografiche o nelle videocamere, oppure come coprocessore all'interno di un PC, destinato all'elaborazione di immagini e di flussi multimediali. In entrambe le configurazioni, il chip esegue un piccolo sistema operativo in tempo reale.

Nexperia svolge tre funzioni: cattura i flussi in entrata e li converte in strutture dati residenti in memoria, le elabora e, infine, le emette in uscita nei formati appropriati ai vari dispositivi collegati. Per esempio, quando un PC è usato come lettore DVD, Nexperia può essere programmato per leggere il flusso video cifrato e compresso proveniente dal disco DVD, decifrarlo, decomprimerlo e quindi inviarlo alla finestra di visualizzazione con la risoluzione appropriata. Una volta caricato il programma di lettura del DVD in Nexperia, tutto ciò può essere svolto in background senza coinvolgere la CPU principale del computer.

Tutti i dati in arrivo vengono innanzitutto caricati in memoria per la loro elaborazione; non c'è nessun collegamento diretto tra i dispositivi in input e quelli in output. La cattura dell'input comprende la decodifica di una gran varietà di risoluzioni e formati video (tra cui MPEG-1, MPEG-2 e MPEG-4), di formati audio (tra cui AAC, Dolby e MP3), cui fa seguito la loro conversione in strutture dati adeguate per la memorizzazione e la successiva elaborazione. L'input può provenire dal bus PCI, da Ethernet o da linee dedicate (per esempio un microfono o un sistema stereo collegato direttamente al chip). Il chip Nexperia ha 456 pin, alcuni dei quali sono disponibili per la connessione diretta di flussi in entrata o in uscita.

L'elaborazione dei dati è affidata al software della CPU TriMedia, che può essere programmata per qualsiasi compito. Tra le elaborazioni più comuni ci sono il deinterlacciamento video per aumentarne la nitidezza, la correzione della luminosità, del contrasto o del colore delle immagini, il loro ridimensionamento, la conversione tra formati video e la riduzione del rumore. Spesso la CPU agisce come il contraente principale di un appalto, subappaltando gran parte del lavoro ai core specializzati del chip.

Le funzionalità output comprendono la codifica delle strutture dati in un formato adeguato al dispositivo di output, la fusione di più sorgenti di dati (video, audio, immagini e grafica 2D) e il controllo dei dispositivi di output. Come per l'input, l'output può essere inviato sul bus PCI, su Ethernet o su linee d'uscita dedicate (per esempio un altoparlante o un amplificatore).

La Figura 8.16 riporta un diagramma a blocchi del chip Nexperia PNX 1500. Esistono altre versioni di Nexperia leggermente diverse, perciò quando ci riferiamo in questo paragrafo a "Nexperia" intendiamo precisamente l'implementazione PNX 1500. Nexperia è composto da quattro sezioni principali: il controllo, l'input, l'elaborazione e l'output. La CPU è il processore VLIW TriMedia a 32 bit e a 300 MHz già trattato nel Paragrafo 8.1.1. Il suo programma, solitamente scritto in C o in C++, ne determina le funzionalità.

Nexperia non ha memoria nel chip, a parte la cache inclusa nella CPU TriMedia, ma la sua interfaccia con la memoria esterna permette di collegare dagli 8 ai 256 MB di SDRAM DDR, quanto basta per gran parte delle applicazioni multimediali. Con un clock di 200 MHz, la larghezza di banda della memoria è di 1,6 GB/s.

Il chip contiene anche un'interfaccia PCI per trasferimenti di 8, 16 o 32 bit a 33 MHz. Quando la CPU è usata come processore autonomo in un dispositivo elettronico di consumo (per esempio in un lettore DVD), l'interfaccia PCI può anche fungere da arbitro del bus. Per esempio, l'interfaccia può essere usata per comunicare con un'unità DVD.

La connessione diretta a Ethernet è fornita da un core dedicato che può gestire connessioni da 10 o 100 Mbps. Una videocamera basata su Nexperia può così inviare a distanza un segnale video digitale tramite l'uscita Ethernet verso un dispositivo di cattura o di visualizzazione.

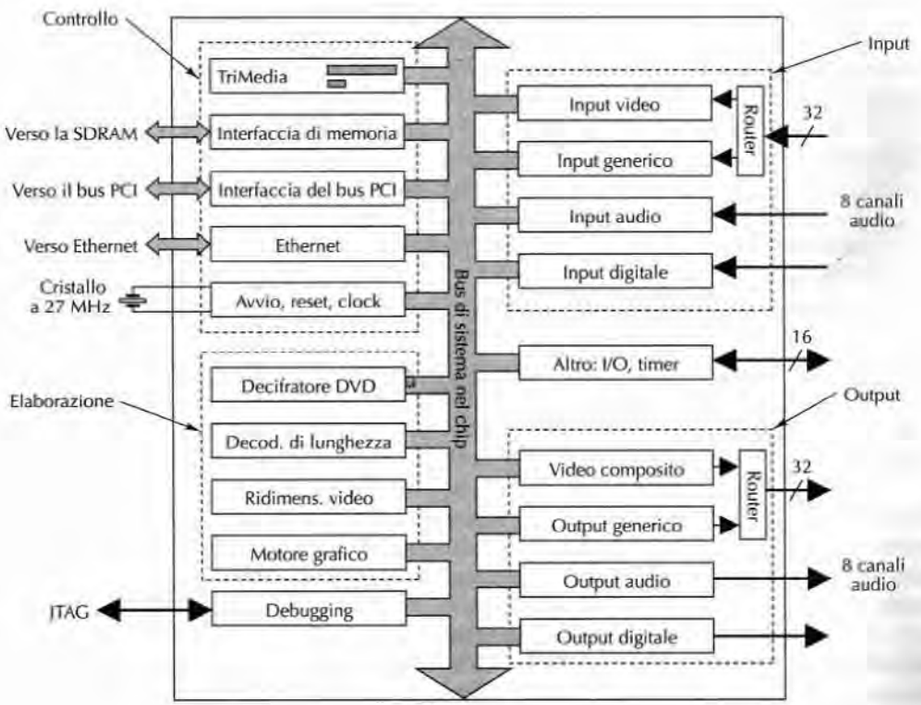


Figura 8.16 Multiprocessore eterogeneo nel chip Nexperia.

Il core successivo gestisce l'avvio del chip (*boot*), il ripristino del suo stato iniziale (*reset*), i clock o altre caratteristiche meno importanti. Se viene inviato un segnale lungo un certo pin di Nexperia, si causa un reset. Il core può essere programmato anche per comportarsi come un interruttore inanimato: se la CPU non riesce ad azionarlo per un certo intervallo di tempo, conclude che il sistema è bloccato e si riavvia di propria iniziativa. Nei dispositivi autonomi il riavvio può essere comandato da una memoria flash.

Il core è temporizzato da un cristallo esterno oscillante a 27 MHz, il cui segnale viene moltiplicato internamente 64 volte per raggiungere gli 1,728 GHz usati nel chip. Di norma la CPU gira a piena velocità e gli altri componenti girano alla velocità necessaria al completamento del proprio lavoro. Tuttavia, la CPU può rallentare il clock per risparmiare energia. È prevista anche una modalità di sospensione per spegnere quasi tutte le funzioni quando non c'è lavoro da svolgere, in modo da risparmiare la carica delle batterie nei dispositivi portatili.

L'unità contiene anche 16 "semafori" utilizzabili per la sincronizzazione tra i dispositivi. Quando un core tenta di scrivere un valore non nullo in un semaforo, la scrittura va a buon fine solo se il semaforo vale 0, altrimenti fallisce e il semaforo resta intatto. La scrittura di 0 ha sempre successo. Poiché un solo core per volta può acquisire il controllo del bus, queste operazioni sono atomiche e forniscono gli strumenti necessari per realizzare la mutua esclusione. Quando vuole acquisire una risorsa, un core cerca di scrivere un valore non nullo in un certo semaforo. Se la scrittura ha successo vuol dire che ha accesso esclusivo alla risorsa finché non la libera scrivendo uno zero nel semaforo corrispondente. Se la scrittura non riesce, il core deve ritentare periodicamente finché la scrittura non va a buon fine, perciò questi semafori non sono proprio i semafori classici definiti nel Capitolo 6.

Volgiamo la nostra attenzione alla sezione di input. Il core di input video riceve un flusso digitale video dell'ampiezza di 10 bit, lo converte in un flusso a 8 bit per mezzo di un algoritmo di *smoothing* ("lisciatura" di un segnale, che risulta in una sfocatura nel caso di immagini) e lo memorizza nella SDRAM esterna. In molti casi, l'input digitale è l'output di un convertitore analogico-digitale esterno, che riceve un segnale televisivo analogico, proveniente da un'antenna o da un cavo.

Il core per input generico è in grado di catturare a 100 MHz un segnale non strutturato e ampio 32 bit, proveniente da una qualsiasi sorgente di dati, e di memorizzarlo nella SDRAM. Può catturare anche dati strutturati per mezzo di marcatori che delineano l'inizio e la fine di record di dati. Il router che si trova davanti ai due input video digitali li demultipla e può anche effettuare alcune trasformazioni video al volo. Il *demultiplexing* è necessario perché vengono usati gli stessi pin per ricevere sia l'input video, sia l'input generico.

Il core dell'input audio può catturare fino a 8 canali di musica o di segnali vocali, con una precisione di 8, 16 o 32 bit e a una frequenza massima di 96 KHz; i segnali sono poi memorizzati nella SDRAM. Può anche decodificare i formati compressi, mischiare i canali, cambiare la frequenza di campionamento e applicare filtri, il tutto eseguito al volo prima di memorizzare i dati audio.

Il core dell'input digitale può ricevere i segnali digitali audio conformi allo standard digitale audio di Sony-Philips (IEC 1937). Grazie a questa caratteristica è possibile trasferire audio digitale da un dispositivo a un altro senza perdita di qualità.

Dopo la cattura dei dati audio, video o di altro tipo, arriva il momento della loro elaborazione, affidata alla sezione successiva del chip. I film venduti o noleggiati su DVD sono

cifrati (*scrambled*) per impedirne la copia. Il decifratore DVD restituisce il filmato originale, compresso con MPEG-2. L'operazione di decifratura opera interamente in memoria: preleva i dati da un buffer e scrive il risultato in un altro buffer.

Il decodificatore di lunghezze prosegue l'elaborazione e rimuove la codifica a lunghezza variabile dai dati compressi con MPEG-2 (o con MPEG-1) producendo dati semi-decompressi a favore del core di elaborazione MPEG, che in questo caso viene sostituito dal software eseguito dal processore TriMedia. La ragione per questa suddivisione dei compiti è che la risoluzione della codifica a lunghezza variabile (decodifica di Huffman e decodifica *run-length*) non riesce a sfruttare appieno le potenzialità di TriMedia. Si è quindi pensato di dedicarle un paio di millimetri quadrati di silicio per svolgerla via hardware. Tutte queste operazioni operano in memoria e producono una semplice mappa di pixel.

La mappa di pixel può essere di tre formati, ciascuno dei quali si diversifica in tre o quattro varianti con dimensioni o parametri diversi. Nel primo formato, detto **a colore indicizzato**, ogni valore della mappa è un indice di una **CLUT** (*Color Look Up Table*, "tabella di ricerca dei colori"), contenente valori di colore a 24 bit, più una **maschera del canale alpha** di 8 bit, usata per determinare il grado di trasparenza quando vengono sovrapposti più livelli. Il secondo formato è **RGB** (alla base del funzionamento degli schermi dei computer), e separa i valori d'intensità dei colori rosso, verde e blu di ogni pixel. Il **YUV** venne ideato per la codifica dei segnali televisivi. Invece di codificare i componenti rosso, verde e blu separatamente, il dispositivo di acquisizione effettua una trasformazione che scompone il segnale in un canale di luminanza (luminosità) e in due canali di crominanza (colore). Questo sistema mette a disposizione una larghezza di banda maggiore per la luminanza rispetto alla crominanza, il che permette di raggiungere una maggiore immunità dal rumore di trasmissione. Il formato YUV è sicuramente appropriato per le applicazioni che ricevono un segnale TV ed emettono ancora un segnale TV. Se si restringe il numero di formati memorizzati a un ventaglio limitato di alternative, è possibile far sì che ogni core sia in grado di leggere l'output di tutti gli altri core.

Il core di ridimensionamento video accetta una lista di compiti di ridimensionamento e li esegue, a un ritmo che può raggiungere i 120 milioni di pixel al secondo. Tra questi compiti ci sono:

1. deinterlacciamento
2. ridimensionamento orizzontale e verticale
3. modifica lineare e non lineare del rapporto larghezza/altezza
4. conversione tra diversi formati di pixel
5. calcolo dell'istogramma di luminanza
6. riduzione dello sfarfallio (*flicker*).

I segnali delle trasmissioni televisive sono **interlacciati** (*interlaced*), ovvero per ogni fotogramma, che consiste in 525 righe di scansione (625 nei sistemi PAL e SECAM), vengono trasmesse prima le righe pari, poi quelle dispari. Il deinterlacciamento permette la **scansione progressiva** che consente una qualità migliore; tutte le righe vengono scandite, elaborate e trasmesse secondo il loro vero ordine e sono quindi aggiornate con frequenza doppia rispetto a quella dei fotogrammi interlacciati (29,97 fotogrammi al secondo per NTSC, 25 per PAL

e SECAM). Il ridimensionamento orizzontale e verticale consente di ingrandire o ridurre le dimensioni delle immagini, per esempio dopo un'operazione di ritaglio. Lo standard televisivo definisce un rapporto larghezza:altezza (*aspect ratio*) di 4:3, ma esistono televisori panoramici da 16:9 che si prestano meglio alla visione di pellicole cinematografiche di rapporto 3:2. Il core di ridimensionamento può convertire questi rapporti, usando algoritmi lineari o non lineari. Può anche convertire tra i formati a colore indicizzato, RGB e YUV e costruire l'istogramma dei valori di luminanza, utile per migliorare la qualità dell'immagine in output. Infine, è possibile effettuare determinate trasformazioni per ridurre il fenomeno di sfarfallio delle immagini.

Il motore grafico effettua il *rendering* ("resa") bidimensionale di un oggetto a partire dalla sua descrizione. Può anche riempire le superfici racchiuse tra i punti descritti ed effettuare le operazioni grafiche di **bitblt**, che consistono nel prendere due mappe rettangolari di pixel e combinarle usando varie funzioni booleane.

Non c'è alcun core per l'elaborazione audio, svolta interamente dal software della CPU TriMedia. I segnali audio richiedono così pochi dati che la loro elaborazione software non costituisce un problema. Inoltre sono poche le applicazioni che richiedono l'elaborazione audio, che si riduce quasi esclusivamente alla conversione tra formati. Il core per il debugging aiuta progettisti e programmatori a effettuare il debugging dell'hardware e del software, e fornisce un'interfaccia agli strumenti **JTAG** (*Joint Test Action Group*), definiti dallo standard IEEE 1149.1.

La sezione di output preleva dalla memoria i dati elaborati e li spedisce in uscita. Il core del video composito riceve una o più strutture dati che rappresentano pixel, le normalizza e le combina, quindi le invia in uscita. I dati in formato indicizzato vengono de-indicizzati al volo per ottenere i pixel reali, e i formati incompatibili vengono opportunamente convertiti. Questo core può anche eseguire la correzione del contrasto, della luminosità o del colore, laddove richiesto, o svolgere l'operazione di *chroma keying*, con cui è possibile isolare dallo sfondo un attore che si staglia davanti a uno schermo blu e collocarlo visivamente su di uno sfondo prelevato da un'altra sorgente. È una tecnica utile per produrre filmati in cui uno o più personaggi in primo piano si muovono davanti a uno sfondo, immobile o che scorre alle loro spalle, diverso da quello di acquisizione. Infine, il risultato viene convertito nel formato appropriato al sistema video o televisivo in questione (per esempio NTSC, PAL o SECAM), compresa la generazione di impulsi per la sincronia orizzontale e verticale.

Ogni sistema basato su Nexperia è in grado di gestire automaticamente, senza costi aggiuntivi, i tre formati televisivi ed è perciò commercializzabile in qualunque parte del mondo. Analogamente, la funzionalità **HDTV** (*High Definition TeleVision*) in un formato qualsiasi può essere ottenuta con una modesta quantità di software aggiuntivo per gestire le conversioni tra le strutture dati in memoria.

Il core di output generico si limita a trasferire a 100 MHz 8, 16 o 32 bit per ogni ciclo, con una larghezza di banda massima di 3,2 Gbps. Se si collega l'output generico di un chip Nexperia all'input generico di un altro chip Nexperia, è possibile trasferire file a un tasso superiore a quello di Ethernet gigabit. Questa interfaccia permette alla CPU di produrre via software output personalizzati di qualsiasi tipo.

Il router di output effettua il *multiplexing* di due sorgenti e mette a disposizione altre funzionalità, come l'aggiornamento a 60 Hz di uno schermo piatto (TFT) di dimensioni

massime di 1280 × 768 pixel, o l'aggiornamento interlacciato o progressivo di apparecchi televisivi. Il multiplexing è reso necessario dal fatto che il video composito e l'output generico condividono gli stessi pin.

Il core di output audio può gestire fino a 8 canali con precisione di 32 bit e a una frequenza di campionamento di 96 KHz. In genere il segnale in uscita è diretto a un convertitore digitale-analogico. L'output digitale può essere connesso all'ingresso digitale di un dispositivo che usa lo standard digitale audio di Sony-Philips.

L'ultimo core gestisce l'I/O d'uso generale, per cui sono disponibili sedici pin, collegabili a bottoni, a interruttori o a LED che permettano la loro attivazione, o la percezione del loro stato, via software. Sono utilizzabili anche per protocolli di rete di media velocità (fino a 20 Mbps) controllati dal software. All'interno di questo core si trovano anche timer, contatori e gestori d'eventi.

Come si è visto, Nexperia presenta un enorme potere di calcolo per applicazioni audiovisive e, come tutti i processori di rete, permette alla CPU di smistare parti ingenti del carico di lavoro. La capacità di calcolo è maggiore di quanto possa sembrare in un primo momento, perché tutti i core girano in parallelo agli altri e alla CPU. Sorprendentemente il processore costa meno di 20 euro se acquistato in grossi quantitativi. Dovrebbe essere ora chiaro quanto siano potenti i coprocessori e specialmente quelli basati su multiprocessori eterogenei nel singolo chip. Per la presentazione di un chip analogo, ma destinato alla telefonia e non alle applicazioni multimediali, si veda (Nickolls et al., 2003).

8.2.3 Crittoprocessori

La sicurezza è il terzo settore in cui sono molto diffusi i coprocessori, soprattutto per quanto riguarda le reti. Quando si stabilisce una connessione tra un client e un server, spesso la prima cosa che viene richiesta a entrambi è l'autenticazione reciproca. A tal fine stabiliscono una connessione cifrata così da poter trasferire i dati in modo sicuro, vanificando ogni tentativo di intromissione nella linea.

Il problema legato alla sicurezza è che, per garantirla, bisogna ricorrere alla crittografia, che richiede però molte risorse di calcolo. Esistono due tipologie di metodi crittografici: **a chiave simmetrica** e **a chiave pubblica**. La prima si basa sull'idea di confondere tutti i bit del messaggio, producendo più o meno ciò che si otterrebbe inserendolo in un frullatore elettronico. La seconda si basa sulla moltiplicazione e sull'elevamento a potenza di numeri molto grandi (per esempio di 1024 bit) e impegna molte risorse di tempo.

Molte aziende hanno prodotto *critto-coprocessori*, spesso sotto forma di schede a innesto nel bus PCI, per gestire i calcoli richiesti dalla cifratura sicura dei dati (per la loro trasmissione o memorizzazione) e dalla loro successiva decifratura. Questi coprocessori sono provvisti di hardware speciale che consente loro di svolgere tutte le necessarie operazioni crittografiche molto più velocemente di quanto possa fare una CPU. Sfortunatamente, la descrizione dettagliata del funzionamento dei crittoprocessori richiederebbe innanzitutto una spiegazione abbastanza accurata della stessa crittografia, il che va oltre gli scopi di questo libro. Si possono trovare maggiori informazioni sui critto-coprocessori in (Daneshbeh e Hasan, 2004; Lutz e Hasan, 2004).

8.3 Multiprocessori con memoria condivisa

Abbiamo visto come introdurre il parallelismo nel chip o all'interno di singoli sistemi tramite l'aggiunta di un coprocessore. Il passo successivo è la combinazione di più CPU per formare sistemi più grandi. Questi sistemi si dividono in due categorie: multiprocessori e multicomputer. Cominciamo con il definire il significato di questi termini, poi ci inoltreremo nell'analisi dei multiprocessori e dei multicomputer.

8.3.1 Multiprocessori e multicomputer a confronto

In un sistema di calcolo parallelo, le varie CPU che operano su parti diverse di uno stesso compito devono poter comunicare tra loro per scambiarsi informazioni. Le modalità della comunicazione sono oggetto di grande dibattito nella comunità degli architetti degli elaboratori. Sono stati proposti e implementati due progetti diversi, i multiprocessori e i multicomputer, che si distinguono per la presenza o per l'assenza di memoria condivisa. Questa differenza influisce sul loro progetto, sulla loro costruzione e programmazione, nonché sui loro costi e dimensioni.

Multiprocessori

Un **multiprocessore** è un calcolatore in cui tutte le CPU condividono una memoria comune, come schematizzato nella Figura 8.17. Tutti i processi che cooperano in un multiprocessore possono condividere un solo spazio degli indirizzi virtuali mappato nella memoria comune. Ogni processo può leggere o scrivere una parola di memoria per mezzo di semplici istruzioni `LOAD` e `STORE`; non serve altro perché del resto si occupa l'hardware. Due processi possono comunicare in modo molto semplice: uno scrive dati in memoria, l'altro è in grado di leggerli.

La facilità con cui due (o più) processi riescono a comunicare tramite operazioni sulla memoria è la ragione della diffusione dei multiprocessori. È un modello semplice da capire per i programmatori ed è applicabile con successo a un gran numero di problemi. Si consideri, per esempio, un programma che ispeziona un'immagine e che elenca tutti gli oggetti in essa contenuti. Una copia dell'immagine è contenuta in memoria, come mostrato dalla Figura 8.17(b). Ciascuna delle 16 CPU elabora un processo singolo cui è stata assegnata una delle 16 porzioni dell'immagine da analizzare. Ciononostante, ogni processo ha accesso all'intera immagine, il che è essenziale poiché alcuni oggetti possono occupare più sezioni contigue. Se un processo scopre che un oggetto si estende all'interno di un'altra porzione, può seguire l'oggetto semplicemente leggendo le parole di quella porzione. In questo esempio alcuni oggetti verranno scoperti da più processi, perciò sarà necessario attribuire delle coordinate che permettano di contare a posteriori quante case, alberi e aeroplani ci sono nell'immagine.

Dal momento che tutte le CPU di un multiprocessore vedono la stessa immagine della memoria, c'è una sola copia del sistema operativo e, di conseguenza, c'è una sola mappa delle pagine e una sola tabella dei processi. Quando un processo si blocca, la sua CPU salva il suo stato nelle tabelle del sistema operativo e cerca al loro interno un altro processo da eseguire. È proprio questa visione di un'unica memoria che distingue un multiprocessore da un multicomputer, in cui invece ogni calcolatore ha la propria copia del sistema operativo.

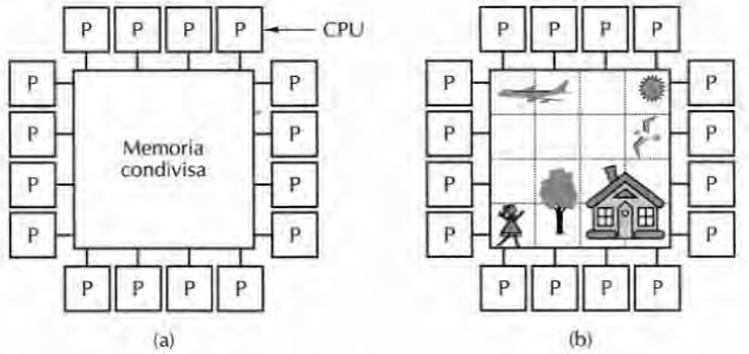


Figura 8.17 (a) Multiprocessore con 16 CPU che condividono una memoria comune. (b) Immagine ripartita in 16 porzioni, ciascuna analizzata da una CPU diversa.

Un multiprocessore, come tutti i calcolatori, deve avere dispositivi di I/O, come i dischi o gli adattatori di rete, e altre periferiche. In alcuni sistemi multiprocessore solo una certa CPU ha accesso ai dispositivi di I/O ed è perciò dotata di funzioni speciali per l'I/O. Invece si parla di **SMP** (*Symmetric MultiProcessor*, "multiprocessore simmetrico") quando tutte le CPU hanno uguale accesso a tutti i moduli di memoria e a tutti i dispositivi di I/O, e sono usate in modo intercambiabile dal sistema operativo.

Multicomputer

Il secondo progetto di architettura parallela prevede per ogni CPU una memoria privata, cioè accessibile solo da essa e o non dalle altre. Un progetto di questo tipo prende il nome di **multicomputer** o di **sistema a memoria distribuita** ed è illustrato nella Figura 8.18(a). L'aspetto fondamentale di un multicomputer, che lo distingue da un multiprocessore, è il fatto che ogni CPU è dotata di una memoria privata e locale, cui può accedere tramite semplici istruzioni `LOAD` e `STORE`, ma cui nessun altra CPU può accedere. Dunque i multiprocessori hanno un solo spazio degli indirizzi fisici comune a tutte le CPU, mentre i multicomputer hanno uno spazio degli indirizzi fisici per ogni CPU.

Visto che le CPU dei multicomputer non possono comunicare attraverso scritture e letture della memoria comune, c'è bisogno di un altro meccanismo di comunicazione: la soluzione adottata è lo scambio di messaggi lungo la rete di interconnessione. Tra i multicomputer annoveriamo BlueGene/L di IBM, Red Storm e i cluster di Google.

L'assenza di una memoria condivisa via hardware ha implicazioni importanti sulla struttura del software di un multicomputer. Non è più possibile avere un solo spazio virtuale degli indirizzi in cui tutti i processi sono in grado di leggere e scrivere per mezzo dell'esecuzione di semplici istruzioni `LOAD` e `STORE`. Per esempio, se la CPU 0 della Figura 8.17 (quella nell'angolo in alto a sinistra) scopre che parte del proprio oggetto si estende all'interno della porzione assegnata alla CPU 1, può comunque accedere alla coda dell'aeroplano continuando a leggere dati dalla memoria. Viceversa, quando la CPU 0 della Figura 8.18 fa la stessa scoperta, non può leggere in alcun modo la memoria della CPU 1. Per accedere a quei dati deve avvalersi di un meccanismo sostanzialmente diverso.

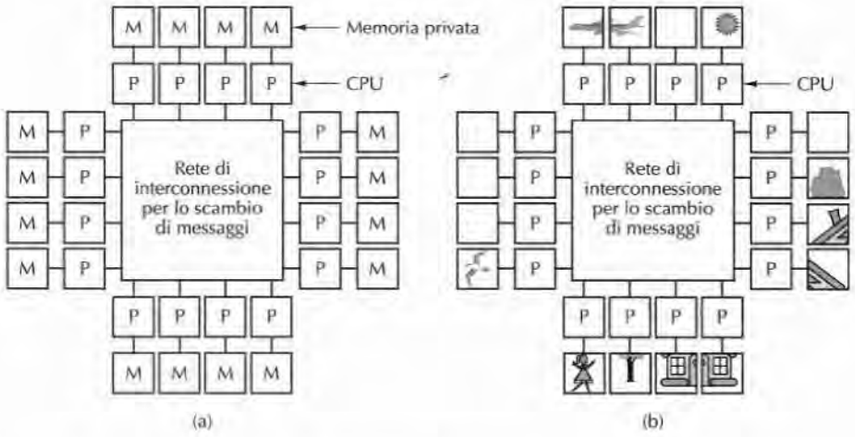


Figura 8.18 (a) Multicomputer con 16 CPU, ciascuna con la propria memoria privata. (b) Immagine bitmap della Figura 8.17 suddivisa tra le 16 memorie.

Infatti deve innanzitutto scoprire (in qualche modo) quale CPU possiede i dati che le interessano e spedirle quindi una richiesta di copia dei dati. Di norma questa operazione blocca la CPU finché la richiesta non viene soddisfatta. All'arrivo del messaggio alla CPU 1, il suo software deve analizzarlo e restituire i dati richiesti. Quando la CPU 0 riceve il messaggio di risposta, il suo software si sblocca e continua l'esecuzione.

La comunicazione tra i processi di un multicomputer avviene quindi tramite le primitive software `send` e `receive`. Ciò rende la struttura del software molto diversa e di gran lunga più complessa rispetto a un multiprocessore. Inoltre è chiaro come diventi particolarmente problematico in un multicomputer riuscire a suddividere i dati correttamente e assegnarli alle locazioni in modo ottimale; la questione non costituiva una grosso problema nei multiprocessori perché in quel caso la collocazione dei dati non influiva sulla correttezza o sulla programmabilità del codice, tutt'al più sulle sue prestazioni. In poche parole, programmare un multicomputer è molto più difficile che programmare un multiprocessore.

Se le cose stanno così, perché mai qualcuno dovrebbe voler costruire multicomputer, quando i multiprocessori sono più facili da programmare? La risposta è semplice: i multicomputer grandi sono molto più semplici ed economici da costruire rispetto a multiprocessori con lo stesso numero di CPU. Riuscire a implementare una memoria condivisa anche solo da qualche centinaia di CPU è praticamente un'impresa, mentre è semplice costruire un multicomputer con 10.000 CPU o più. Nel seguito del capitolo studieremo un multicomputer con più di 50.000 CPU.

Ci troviamo davanti a un dilemma: i multiprocessori sono difficili da costruire, ma facili da programmare, i multicomputer sono facili da costruire, ma difficili da programmare. Per queste ragioni sono stati compiuti sforzi ingenti per riuscire a costruire sistemi ibridi che fossero relativamente facili da costruire e programmare. Come risultato di questi sforzi si è giunti alla conclusione che la memoria condivisa può essere implementata in modi diversi,

ciascuno con i propri vantaggi e svantaggi. Così una fetta considerevole della ricerca attuale nelle architetture parallele si concentra sulla convergenza di architetture multiprocessore e multicomputer che conservi i punti di forza di ciascuna. Il Santo Graal di questa disciplina è riuscire a ideare progetti **scalabili**, ovvero che continuino a comportarsi bene anche al crescere delle dimensioni per l'aggiunta di altre CPU.

Un approccio per la costruzione di sistemi ibridi si basa sul fatto che i sistemi di calcolo moderni non sono monolitici, bensì costruiti come una serie di livelli (il tema di questo libro). Questa intuizione apre la strada all'implementazione della memoria condivisa a livelli diversi, come mostra la Figura 8.19. Nella Figura 8.19(a) la memoria condivisa è implementata dall'hardware come in un vero multiprocessore. Secondo questo progetto, c'è una sola copia del sistema operativo con un solo insieme di tabelle e, in particolare, con una sola tabella di allocazione della memoria. Quando un processo ha bisogno di più memoria, invia una trap al sistema operativo e questo cerca una pagina libera nella propria tabella e la mappa nello spazio degli indirizzi del chiamante. Dal punto di vista del sistema operativo c'è una sola memoria ed esso si limita a mantenere traccia via software dell'attribuzione delle pagine ai processi. Come vedremo, sono possibili molte implementazioni hardware di una memoria condivisa.

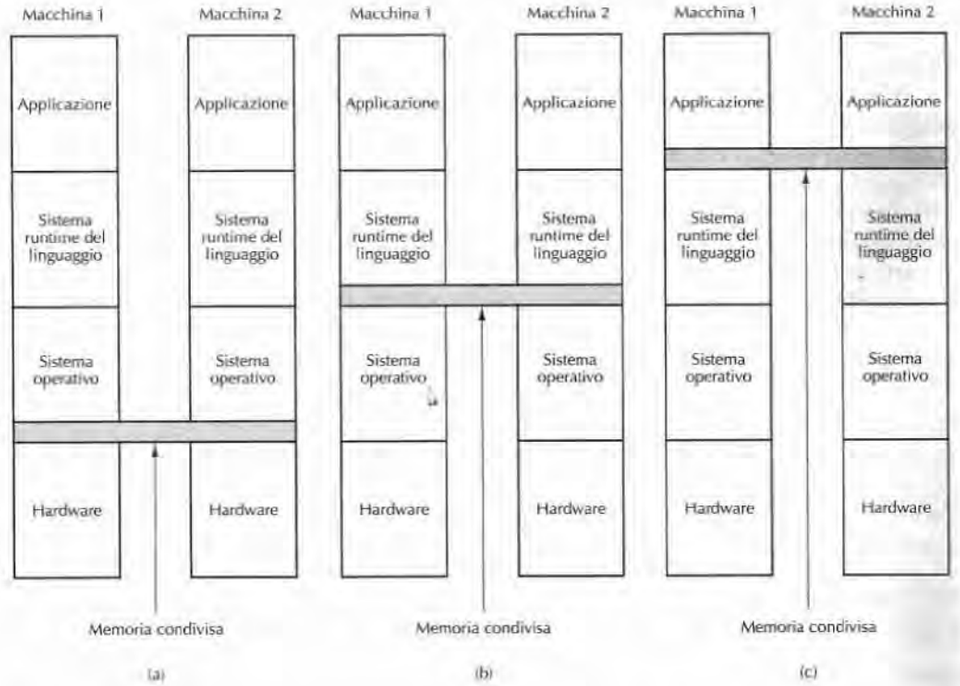


Figura 8.19 Vari livelli d'implementazione di una memoria condivisa. (a) Nell'hardware. (b) Nel sistema operativo. (c) Nel sistema runtime del linguaggio.

Una seconda possibilità è usare l'hardware di un multicomputer e servirsi del sistema operativo per simulare la memoria condivisa come uno spazio paginato degli indirizzi virtuali condiviso in tutto il sistema. Questo approccio si chiama **DSM** (*Distributed Shared Memory*, "memoria condivisa distribuita"; Li e Hudak, 1989) e prevede che ogni pagina sia localizzata in una delle memorie della Figura 8.18(a). Ogni macchina ha una propria memoria virtuale e una propria tabella delle pagine. Quando una CPU effettua una LOAD o una STORE all'interno di una pagina di cui non dispone, si verifica una trap rivolta al sistema operativo; questo localizza la pagina e richiede alla CPU che la possiede correntemente di estrometterla dalla memoria e di spedirla lungo la rete di interconnessione. Una volta recapitata, la pagina viene mappata in memoria della prima CPU e l'istruzione può riprendere la propria esecuzione. A tutti gli effetti, il sistema operativo non sta facendo altro che gestire gli errori di pagina attingendo a una memoria distante invece che al disco. Dal punto di vista dell'utente, il sistema sembra un sistema a memoria virtuale. Analizzeremo DSM nel prosieguo del capitolo.

Una terza possibilità è implementare la memoria condivisa all'interno del sistema runtime (eventualmente specifico di un linguaggio) nel livello utente. Secondo questo approccio, il linguaggio di programmazione fornisce una specie di astrazione della memoria condivisa che viene poi implementata dal compilatore e dal sistema runtime. Per esempio, il modello Linda si basa sull'astrazione di uno spazio condiviso di tuple (record di dati contenenti un insieme di campi). I processi di tutte le macchine possono prelevare una tupla dallo spazio condiviso delle tuple o inserirvene una. Poiché il controllo degli accessi è effettuato tutto via software (dal sistema runtime di Linda) non c'è bisogno di hardware speciale o di supporto da parte del sistema operativo.

Un altro esempio di memoria condivisa implementata dal sistema runtime di un linguaggio è il modello Orca di condivisione degli oggetti di dati. In Orca, i processi condividono oggetti generici e non tuple, e possono eseguire su di loro alcuni metodi specifici degli oggetti stessi. Quando un metodo modifica lo stato interno di un oggetto, spetta al sistema runtime assicurarsi che tutte le copie dell'oggetto residenti sulle diverse macchine vengano aggiornate simultaneamente. Sottolineiamo che non c'è bisogno di alcuna assistenza da parte dell'hardware o del sistema operativo perché gli oggetti sono un concetto puramente software. In questo capitolo ci occuperemo ancora di Linda e di Orca.

Tassonomia dei calcolatori paralleli

Torniamo ora al nostro argomento principale: l'architettura dei calcolatori paralleli. Nel corso degli anni sono stati proposti e costruiti molti tipi di calcolatori paralleli, dunque viene naturale chiedersi se è possibile organizzarli in una tassonomia di categorie. Molti ricercatori si sono dedicati a questa attività, con risultati diversi (Flynn, 1972; Treleaven, 1985). Purtroppo non è ancora nato il Linneo[3] del calcolo parallelo. L'unico schema abbastanza usato è quello di Flynn (Figura 8.20), che resta comunque molto approssimativo.

La classificazione di Flynn si basa su due concetti: i flussi d'istruzioni e i flussi di dati. Un flusso d'istruzioni corrisponde a un program counter. Un sistema con n CPU ha n program counter, quindi n flussi d'istruzioni.

[3] Carlo Linneo (1707-1778): biologo svedese che inventò il sistema di classificazione delle piante e degli animali ancora in uso e che cataloga gli individui in base alle categorie di regno, phylum (tipo o divisione), classe, ordine, famiglia, genere e specie.

Flussi d'istruzioni	Flussi di dati	Nome	Esempi
1	1	SISD	Modello di Von Neumann
1	Molteplici	SIMD	Supercomputer vettoriali, unità di calcolo vettoriali
Molteplici	1	MISD	Forse nessuno
Molteplici	Molteplici	MIMD	Multiprocessori, multicomputer

Figura 8.20 Tassonomia di Flynn dei calcolatori paralleli.

Un flusso di dati è un insieme di operandi. L'esempio del calcolo della temperatura del Paragrafo 1.3.3 è un caso in cui ci sono molteplici flussi di dati, uno per ogni sensore.

I flussi d'istruzioni e di dati sono in buona parte indipendenti, perciò si danno le quattro possibili combinazioni della Figura 8.20. SISD è la macchina sequenziale classica di Von Neumann, caratterizzata da un solo flusso d'istruzioni, un solo flusso di dati e svolge una sola attività alla volta. Le macchine SIMD hanno una sola unità di controllo che emette un'istruzione alla volta, ma dispongono di varie ALU per operare simultaneamente su diversi insiemi di dati. L'ILLIAC IV (Figura 2.7) è il prototipo di macchina SIMD. I mainframe SIMD sono sempre più rari, sebbene i calcolatori convenzionali possano avere alcune istruzioni SIMD per l'elaborazione di materiale audiovisivo (per esempio le istruzioni SSE del Pentium). Comunque sia, c'è un nuovo settore in cui sembrano giocare un ruolo importante le idee del paradigma SIMD: gli *stream processor*. Si tratta di macchine progettate specificatamente per la gestione del rendering multimediale e potrebbero acquisire in futuro una certa importanza (Kapasi et al., 2003).

Le macchine MISD fanno parte di una categoria un po' strana, in cui ci sono molteplici istruzioni che operano sugli stessi dati. Non è ben chiaro se esista una qualche macchina di questo tipo, anche se alcuni ritengono che le macchine a pipeline siano macchine MISD.

Infine vengono le macchine MIMD, semplicemente un insieme di CPU indipendenti che operano come componenti di un sistema più grande. Molti dei calcolatori paralleli fanno parte di questa categoria. I multiprocessori e i multicomputer sono macchine MIMD.

La tassonomia di Flynn si ferma qui, ma l'abbiamo arricchita come mostrato nella Figura 8.21. Abbiamo suddiviso le macchine SIMD in due sottogruppi: il primo comprende i supercomputer numerici e le macchine che operano su vettori, effettuando la stessa operazione su ciascun elemento del vettore; il secondo contiene le macchine di tipo parallelo, come l'ILLIAC IV, in cui un'unità principale di controllo invia le istruzioni a molte ALU indipendenti.

Nella nostra tassonomia abbiamo diviso la categoria MIMD in multiprocessori (macchine a condivisione di memoria) e multicomputer (macchine a scambio di messaggi). Esistono tre tipi di multiprocessori, caratterizzati dal modo in cui implementano la memoria condivisa; sono chiamati **UMA** (*Uniform Memory Access*, "accesso uniforme alla memoria"), **NUMA** (*NonUniform Memory Access*) e **COMA** (*Cache Only Memory Access*). Queste sottocategorie esistono perché in genere la memoria dei multiprocessori molto grandi è divisa in più moduli. Le macchine UMA hanno la proprietà di garantire lo stesso tempo d'accesso a ogni modulo di memoria da parte di ogni CPU, ovvero ogni parola di memoria può

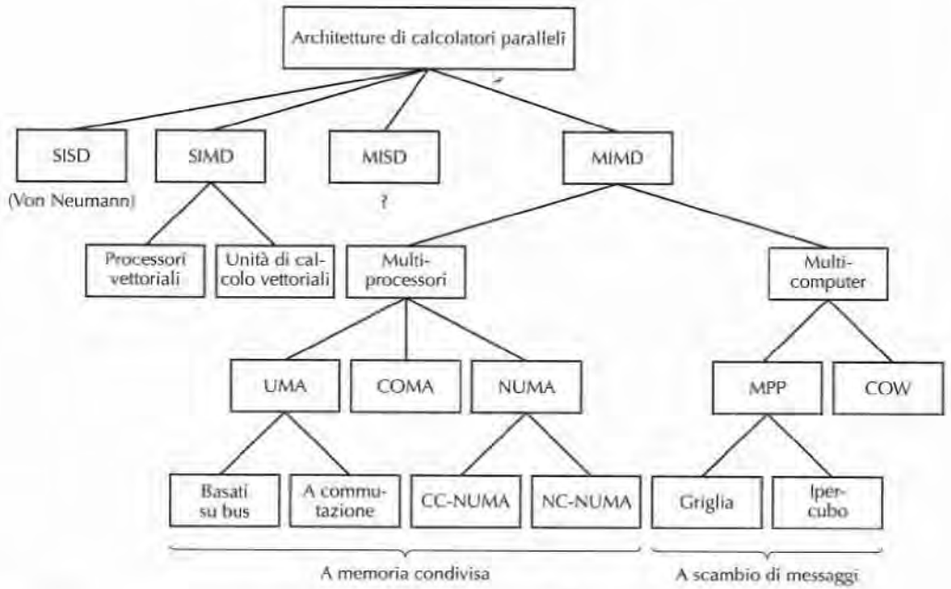


Figura 8.21 Tassonomia dei calcolatori paralleli.

essere letta tanto velocemente quanto qualsiasi altra. Se questa proprietà è tecnicamente impossibile da garantire, gli accessi più veloci sono rallentati alla velocità dei più lenti così che i programmatori non si accorgano della differenza. È questa l'accezione di "uniforme" che si intende in questo contesto. L'uniformità permette di fare previsioni sulle prestazioni, il che è importante per la scrittura di codice efficiente.

Viceversa, nei multiprocessori NUMA questa proprietà non vale. Spesso c'è un modulo di memoria vicino a ogni CPU e il suo accesso è molto più veloce rispetto ai moduli distanti. Perciò, ai fini delle prestazioni, conta molto la collocazione dei dati e dei programmi. Anche le macchine COMA non sono uniformi, ma in un modo diverso. Ci occuperemo più tardi di queste categorie di macchine.

L'altra categoria principale delle macchine MIMD è costituita dai multicomputer che, a differenza dei multiprocessori, non hanno una memoria principale condivisa a livello di architettura. In altre parole, il sistema operativo di una CPU di un multicomputer non può accedere alla memoria di un'altra CPU tramite la sola esecuzione di un'istruzione `LOAD`. Deve invece spedire un messaggio e aspettare la relativa risposta. La capacità del sistema operativo di leggere parole a distanza effettuando una semplice `LOAD` è ciò che distingue i multiprocessori dai multicomputer. Abbiamo già detto che anche i programmi utente di un multicomputer possono essere resi in grado di accedere alle memorie distanti tramite istruzioni `LOAD` e `STORE`, però si tratta in questo caso di un'illusione di cui si fa carico il sistema operativo e che non è sorretta dall'hardware. Si tratta di una differenza impercettibile, eppure

molto importante. Per queste ragioni i multicomputer sono detti anche **NORMA** (*NO Remote Memory Access*).

È possibile individuare due categorie di multicomputer. La prima contiene processori di tipo **MPP** (*Massively Parallel Processor*), supercomputer molto costosi costituiti da molte CPU strettamente legate da una rete d'interconnessione ad alta velocità brevettata dal costruttore. L'IBM SP/3 ne è un esempio commerciale molto noto.

L'altra categoria raggruppa i PC consueti o le workstation, eventualmente montati in scaffali (*rack*), collegati tramite una tecnologia commerciale d'interconnessione pronta per l'uso. Dal punto di vista teorico non c'è molta differenza, ma i supercomputer giganti sono usati per scopi diversi rispetto alle reti di PC, assemblate dagli utenti a un costo che rappresenta una frazione molto piccola rispetto ai milioni di euro necessari all'acquisto di un MPP. Queste macchine fatte in casa vengono denominate in modi diversi, tra cui **NOW** (*Network of Workstations*), **COW** (*Cluster Of Workstations*) o semplicemente **cluster**.

8.3.2 Semantica della memoria

Anche se i multiprocessori mostrano la memoria alle CPU come un unico spazio condiviso degli indirizzi, spesso ci sono diversi moduli di memoria, ciascuno contenente una parte della memoria fisica. CPU e memorie sono collegate tramite una complessa rete d'interconnessione, come già illustrato nel Paragrafo 8.1.2. È possibile che diverse CPU tentino di leggere contemporaneamente la stessa parola di memoria, mentre altre CPU cercano di sovrascriverne il contenuto; ed è anche possibile che i messaggi di richiesta si sorpassino l'un l'altro durante il tragitto e che vengano quindi consegnati in un ordine diverso rispetto alla loro emissione. Si aggiunga poi il problema dell'esistenza di copie multiple di alcuni blocchi di memoria (per esempio nelle cache) e si capirà facilmente che è necessario adottare delle regole precise per prevenire il caos totale. In questo paragrafo vedremo che cosa voglia dire davvero condividere la memoria e come sia possibile agire nelle circostanze suddette.

Un modo per concepire la semantica della memoria è come contratto tra l'hardware della memoria e il software (Adve e Hill, 1990): se il software accetta di stare ad alcune regole, la memoria prende l'impegno di fornire certi risultati. L'analisi si sposta quindi sulla determinazione di queste regole, chiamate **modelli di consistenza**, per cui esistono molte proposte e altrettante implementazioni.

Per chiarire la natura del problema, ipotizziamo che la CPU 0 scriva il valore 1 in una certa parola di memoria e che poco dopo la CPU 1 scriva il valore 2 nella stessa locazione. A questo punto la CPU 2 legge la parola e ottiene il valore 1. Che cosa dovrebbe fare il proprietario di quel calcolatore, riportarlo in negozio per una riparazione?

Consistenza stretta

Il modello più semplice è la **consistenza stretta** (*strict consistency*), secondo cui ogni lettura di una locazione *x* deve sempre restituire il valore della scrittura più recente. I programmatori adorano questo modello, che però è pressoché impossibile da implementare se non con un solo modulo di memoria che serva tutte le richieste nell'ordine in cui sono recapitate, senza effettuare caching né duplicazione di dati. Sfortunatamente questa implementazione renderebbe la memoria un collo di bottiglia per l'intero sistema e perciò non va presa in seria considerazione.

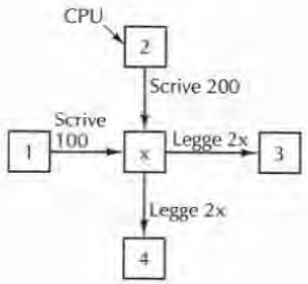
Consistenza sequenziale

Il secondo modello più promettente è la **consistenza sequenziale** (*sequential consistency*; Lamport, 1979). L'idea è la seguente: se ci sono più richieste di lettura e scrittura per una locazione, l'hardware effettua un rimescolamento (non deterministico) dell'ordine delle richieste e tutte le CPU vedono lo stesso ordine.

Esaminiamone il significato con un esempio. Si ipotizzi che la CPU 1 scriva il valore 100 nella parola x e 1 ns dopo la CPU 2 scriva il valore 200 nella stessa parola x. Quando è passato 1 ns dall'emissione della seconda scrittura (ma non necessariamente dal suo completamento) altre due CPU, la 3 e la 4, effettuano entrambe due letture di x in rapida successione, come mostrato dalla Figura 8.22(a). Le Figure 8.22(b)-(d) mostrano tre possibili ordinamenti dei sei eventi (due scritture e sei letture): nella prima figura la CPU 3 ottiene (200, 200) e la CPU 4 ottiene (200, 200); nella Figura 8.22(c) ottengono rispettivamente (100, 200) e (200, 200); nella Figura 8.22(d) ottengono rispettivamente (100, 100) e (200, 100). Sono tutte combinazioni lecite, come lo sono le altre che non abbiamo elencato.

Comunque vada, è impossibile che la CPU 3 ottenga (100, 200) mentre la CPU 4 ottiene (200, 100), e questa è l'essenza stessa della consistenza sequenziale. Se ciò potesse capitare, vorrebbe dire che, secondo la CPU 3, la scrittura di 100 da parte della CPU 1 è stata completata prima della scrittura di 200 da parte della CPU 2, e questo va bene. Però vorrebbe dire anche che, secondo la CPU 4, la scrittura di 200 da parte della CPU 2 è stata completata prima della scrittura di 100 da parte della CPU 1. Anche questo risultato è possibile in sé, ma il fatto è che la consistenza sequenziale garantisce l'esistenza di un unico ordinamento globale delle scritture visibile a tutte le CPU. Se la CPU 3 osserva per prima la scrittura di 100, così deve essere anche per la CPU 4.

Nonostante la consistenza sequenziale sia una regola meno potente della consistenza stretta, si rivela comunque molto utile. Infatti stabilisce che, quando più eventi si verificano in modo concorrente, esiste un ordine secondo cui avvengono, determinato forse dalla temporizzazione o dal caso, ma si tratta pur sempre di un ordine che deve essere osservato da tutti i processori. Benché questa osservazione appaia ovvia, passiamo ora a modelli di consistenza che non garantiscono neanche questo.



(a)

(b)	(c)	(d)
W100	W100	W200
W200	R3 = 100	R4 = 200
R3 = 200	W200	W100
R3 = 200	R4 = 200	R3 = 100
R4 = 200	R3 = 200	R4 = 100
R4 = 200	R4 = 200	R3 = 100

Figura 8.22 (a) Due CPU che scrivono una parola di memoria letta da altre due CPU. (b)-(d) Tre permutazioni dell'ordine temporale delle due scritture e delle quattro letture.

Consistenza di processore

La **consistenza di processore** (*processor consistency*; Goodman, 1989) è un modello di consistenza meno rigido, ma più facile da implementare su multiprocessori grandi. Manifesta due proprietà:

1. le scritture effettuate da qualsiasi CPU sono percepite da tutte le altre CPU nello stesso ordine in cui sono state emesse
2. per ogni parola di memoria, tutte le CPU vedono lo stesso ordine di scritture al suo interno.

La prima di queste due importanti proprietà afferma che, se la CPU 1 emette in sequenza le scritture di valori 1A, 1B e 1C in una locazione di memoria, tutti gli altri processori le vedono in quello stesso ordine. Ciò equivale a dire che, ogni processore che osservi il valore assunto da quella locazione di memoria mediante un ciclo di letture, non potrà mai ritenere il valore 1B scritto prima del valore 1A, e così via. La seconda proprietà serve affinché ogni parola di memoria contenga un valore non ambiguo al termine di numerose scritture da parte di più CPU. Tutti devono concordare su chi ha scritto per ultimo.

Pur con questi vincoli, al progettista resta ampio margine di scelta. Si consideri che cosa succede se la CPU 2 emette le scritture 2A, 2B e 2C concorrentemente alle tre scritture della CPU 1. Le altre CPU che stanno osservando la memoria mediante letture ripetute, vedranno un rimescolamento delle sei scritture, per esempio 1A, 1B, 2A, 2B, 1C, 2C oppure 2A, 1A, 2B, 2C, 1B, 1C o altri ancora. La consistenza di processore *non* garantisce che tutte le CPU vedano lo stesso ordinamento (a differenza della consistenza sequenziale). Perciò è del tutto legittimo per l'hardware comportarsi in questo modo e mostrare a ogni CPU un ordinamento potenzialmente diverso. Ciò che *è* garantito invece è che nessuna CPU veda una sequenza in cui 1B viene prima di 1A, e così via. L'ordine relativo delle scritture emesse da ogni singola CPU viene mantenuto comunque.

Va detto che alcuni autori definiscono la consistenza di processore in modo diverso e che non richiedono la seconda proprietà.

Consistenza debole

Il modello successivo, la **consistenza debole** (*weak consistency*), non garantisce nemmeno l'ordinamento delle scritture effettuate da una sola CPU (Dubois et al., 1986). In una memoria debolmente consistente una CPU potrebbe vedere 1A prima di 1B e un'altra potrebbe vedere l'esatto contrario. Tuttavia, per riportare un po' di ordine nel caos, le memorie debolmente consistenti sono dotate di variabili e di operazioni di sincronizzazione. Quando viene eseguita una sincronizzazione, vengono completate tutte le scritture pendenti e non è possibile effettuarne di nuove finché non termina la sincronizzazione. La sincronizzazione effettua un "flush della pipeline" (scarica le operazioni in coda) e conduce la memoria a uno stato stabile, senza operazioni pendenti. Le operazioni di sincronizzazione sono esse stesse sequenzialmente consistenti, cioè anche se vengono emesse da più CPU, tutti i processori percepiscono lo stesso ordine globale della loro esecuzione.

In questo modello di consistenza, il tempo è diviso in epoche ben definite e delimitate dalle sincronizzazioni (sequenzialmente consistenti), come illustrato dalla Figura 8.23. Non è garantito l'ordine relativo tra 1A e 1B, cioè CPU diverse potrebbero osservare le due scrit-

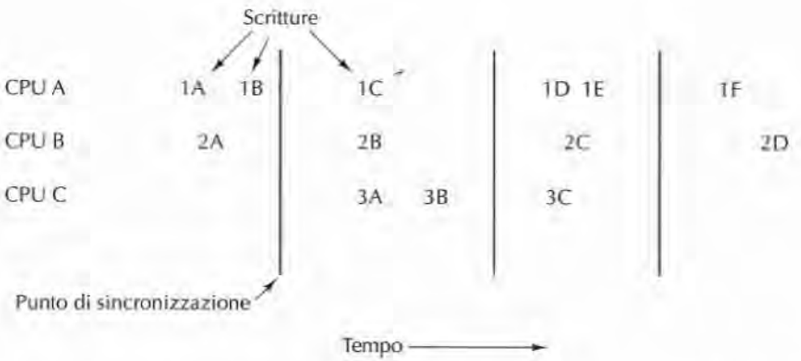


Figura 8.23 La memoria a consistenza debole utilizza la sincronizzazione per suddividere il tempo in una successione di epoche.

ture in ordine diverso (una CPU potrebbe vedere 1A, 1B, un'altra potrebbe vedere 1B, 1A). Però tutte le CPU devono vedere 1B prima di 1C, perché l'operazione di sincronizzazione forza il completamento di 1A, 1B e 2A prima che 1C, 2B, 3A o 3B possano cominciare. La sincronizzazione è un modo con cui il software può imporre un certo ordinamento nella sequenza di eventi, ma al prezzo di un flush della pipeline di memoria, che richiede un certo tempo.

Consistenza dopo rilascio

La consistenza debole è abbastanza inefficiente perché comporta il completamento di tutte le operazioni di memoria pendenti e il trattenimento di tutte le operazioni nuove finché non vengano terminate le precedenti. La **consistenza dopo rilascio** (*release consistency*) costituisce un miglioramento e adotta un modello simile alle sezioni critiche (Gharachorloo et al., 1990). L'idea alla base di questo modello è la seguente: quando un processo esce da una regione critica, non è necessario richiedere il completamento immediato di tutte le scritture. Basta solo assicurarsi che vengano effettuate prima del reingresso di un altro processo nella regione critica.

Secondo questo modello, l'operazione di sincronizzazione fornita dalla consistenza debole viene suddivisa in due operazioni differenti. Per leggere o scrivere una variabile condivisa, la CPU (cioè il suo software) deve prima eseguire un'operazione `acquire` sulla variabile di sincronizzazione per acquisire l'accesso esclusivo ai dati condivisi. Quindi la CPU può usarli come vuole, sia in lettura sia in scrittura. Al termine della sua attività, la CPU invoca l'operazione `release` sulla variabile di sincronizzazione per indicare il suo rilascio. La `release` non forza il completamento di tutte le scritture pendenti, bensì attende per il proprio completamento che siano avvenute tutte le scritture emesse precedentemente. Inoltre non ritarda l'avvio di nuove operazioni di memoria.

All'emissione di una `acquire`, per prima cosa si verifica se tutte le `release` precedenti sono state completate. Se così non è, la `acquire` resta sospesa fino al loro termine (dunque finché non sono state completate tutte le scritture precedenti). Così facendo, se una `acquire` è abbastanza lontana dalla `release` precedente, non deve aspettare alcunché prima

di entrare nella regione critica. Se invece avviene troppo presto, dovrà essere ritardata fino al completamento di tutte le `release` pendenti, per garantire il corretto aggiornamento delle variabili della sezione critica. Si tratta di uno schema leggermente più complesso della consistenza debole, ma presenta il vantaggio significativo di non ritardare le istruzioni così spesso, pur mantenendo la consistenza.

La consistenza di memoria non è un capitolo chiuso, la ricerca di nuovi modelli va avanti (Cain e Lipasti, 2004; Hammond et al., 2004).

8.3.3 Architetture di multiprocessori simmetrici UMA

I multiprocessori più semplici si basano su un solo bus, come illustrato nella Figura 8.24(a). Ci sono due o più CPU, uno o più moduli di memoria, e tutti usano lo stesso bus per la comunicazione. Quando una CPU vuole leggere una parola di memoria, per prima cosa verifica se il bus è occupato; se non lo è, può inserire nel bus l'indirizzo della parola voluta, attivare qualche segnale di controllo e aspettare finché la memoria non spedisce sul bus la parola desiderata.

Se invece il bus è occupato, la CPU aspetta che si liberi; sta proprio qui il problema di questo progetto. Se le CPU sono due o tre, la contesa del bus è ancora gestibile; se sono 32 o 64 allora diventa intrattabile perché il sistema sarà limitato completamente dalla larghezza di banda del bus, e la maggior parte delle CPU resterà inattiva per gran parte del tempo.

La soluzione del problema è l'aggiunta di una cache a ogni CPU, come rappresentato nella Figura 8.24(b). La cache può trovarsi all'interno del chip della CPU, vicino alla CPU, sulla scheda del processore o in una posizione intermedia. La cache locale consente di soddisfare molte più letture attingendo al proprio interno, perciò ci sarà molto meno traffico sul bus e il sistema potrà gestire più CPU. Il caching risulta qui una scelta vincente.

Una possibilità alternativa è il progetto della Figura 8.24(c), in cui ogni CPU dispone non solo di una cache, ma anche di una memoria locale privata, cui può accedere grazie a un bus dedicato (privato). Per sfruttare questa configurazione in modo ottimale, il compilatore dovrebbe usare la memoria privata per contenere i sorgenti del programma, le stringhe, le costanti e gli altri dati in sola lettura, gli stack e le variabili locali. La memoria condivisa verrebbe usata solo per la scrittura delle variabili condivise. Questa collocazione riduce in molti casi il traffico sul bus, ma richiede la cooperazione attiva da parte del compilatore.

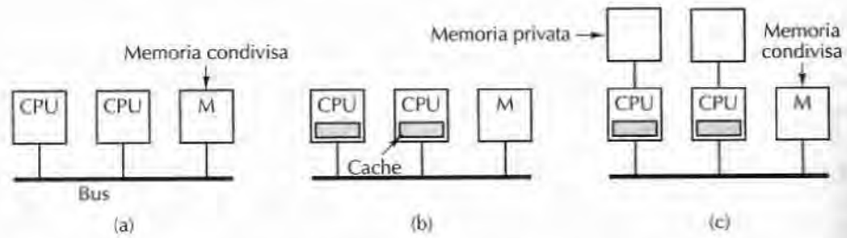


Figura 8.24 Tre multiprocessori basati su bus. (a) Senza caching. (b) Con caching. (c) Con caching e memorie private.

Cache snooping

Il ragionamento fatto circa le prestazioni è corretto, ma abbiamo liquidato troppo velocemente una questione fondamentale. Se la memoria ha consistenza sequenziale, che cosa succede nel caso in cui la CPU 1 ha nella propria cache una linea e la CPU 2 cerca di leggere una parola dalla stessa linea di cache? In assenza di regole speciali, quest'ultima non farebbe altro che procurarsi una copia della linea da inserire nella propria cache. In teoria è ammissibile che una linea sia presente in due cache diverse. Se poi la CPU 1 modifica la linea e, immediatamente dopo, la CPU 2 legge la propria copia, ma si troverà a manipolare **dati scaduti** (*stale data*), violando così il contratto tra software e memoria. Il programma in esecuzione sulla CPU 2 viene danneggiato da questo comportamento.

A tale proposito la letteratura specializzata parla del **problema della coerenza** o **consistenza della cache**. Si tratta di un problema molto grave che, se non risolto, preclude l'uso del caching e i multiprocessori orientati al bus sarebbero così limitati a due o tre CPU. In ragione dell'importanza della questione, sono state proposte negli anni molte soluzioni (per esempio Goodman, 1983; Papamarcos e Patel, 1984) che definiscono un **protocollo di coerenza delle cache** e, sebbene differiscano nei dettagli, impediscono la presenza simultanea di versioni discordanti di una stessa linea all'interno di due o più cache. L'unità di trasferimento e di memorizzazione di una cache si dice *linea di cache* ed è solitamente di 32 o 64 byte.

Tutte le soluzioni richiedono controllori di cache progettati per essere in grado di "origliare" sul bus, cioè di monitorare tutte le richieste che transitano sul bus e che provengono da altre CPU o cache, al fine di intraprendere le azioni opportune. Questi dispositivi si dicono **snooping cache** o **snoopy cache**, cioè cache che letteralmente *ficcano il naso* nel bus. Il protocollo di coerenza delle cache è composto dall'insieme di regole osservate dalle cache, dalla CPU e dalla memoria per prevenire la presenza di versioni diverse dei dati all'interno di varie cache.

Il più semplice protocollo di coerenza delle cache si chiama **write through** e può essere capito facilmente attraverso le quattro situazioni elencate nella Figura 8.25. Quando una CPU tenta di leggere una parola che non si trova nella sua cache, evento detto *read miss*, il controllore carica nella cache la linea contenente quella parola. La linea proviene dalla memoria che, secondo questo protocollo, viene mantenuta costantemente aggiornata. Le successive richieste di letture possono essere soddisfatte dalla cache (*read hit*).

In seguito a un fallimento in scrittura, evento detto *write miss*, la parola modificata viene scritta in memoria principale, mentre la linea contenente la parola *non* viene caricata nella cache. In caso di successo in scrittura (*write hit*) viene aggiornata la cache e inoltre la parola viene scritta direttamente in memoria principale. La caratteristica più importante di questo protocollo è che tutte le operazioni di scrittura agiscono sulla memoria, che risulta perciò sempre aggiornata.

Riconsideriamo tali azioni, questa volta dal punto di vista di una *snooping cache* (colonna più a destra nella Figura 8.25). Chiamiamo cache 1 la cache che effettua le azioni e cache 2 la cache "ficcanaso". Dopo un read miss, la cache 1 effettua una richiesta di caricamento di una linea dalla memoria; la cache 2 osserva questa richiesta senza intervenire. Dopo un read hit la richiesta viene soddisfatta localmente, perciò non viene inviata alcuna richiesta sul bus, quindi la cache 2 non sa della lettura della cache 1.

Azione	Richiesta locale	Richiesta a distanza
Read miss	Preleva dati in memoria	Nessuna
Read hit	Usa dati dalla cache locale	"
Write miss	Aggiorna i dati in memoria	"
Write hit	Aggiorna la cache e la memoria	Invalida l'elemento di cache

Figura 8.25 Protocollo write-through di coerenza delle cache.

Le situazioni in scrittura sono più interessanti. Se la CPU 1 effettua una scrittura, la cache 1 invia una richiesta di scrittura sul bus, sia in caso di miss sia di hit. A ogni scrittura, la cache 2 verifica se possiede la parola che deve essere scritta. In caso negativo, dal suo punto di vista si tratta di una semplice richiesta a distanza che provoca un write miss, perciò non intraprende alcuna azione (un fallimento a distanza nella Figura 8.25 significa che la parola non è presente nella snooping cache, indipendentemente dalla sua eventuale presenza nella cache 1; una richiesta può causare localmente un hit nella prima cache e un miss nella snooping cache, o viceversa).

Se invece la cache 1 scrive una parola presente nella cache 2 (richiesta a distanza che ha successo), quest'ultima deve intraprendere qualche azione per evitare di avere dati scaduti, perciò contrassegna come non valido l'elemento di cache contenente la parola modificata. Ciò equivale a rimuovere l'elemento dalla cache. Poiché tutte le cache monitorano tutte le richieste di bus, ogni scrittura di una parola provoca l'aggiornamento della cache del richiedente e della memoria, la rimozione dei dati scaduti dalle altre cache, e così si previene l'insorgenza di versioni incoerenti.

Naturalmente è possibile che la CPU della cache 2 voglia leggere quella stessa parola al ciclo successivo. In tal caso, la cache 2 leggerebbe la parola dalla memoria, che è perfettamente aggiornata. Dopo la lettura la cache 1, la cache 2 e la memoria avrebbero tutte una copia fedele del dato. Se una delle due CPU effettuasse ora una scrittura, l'elemento di cache dell'altra CPU verrebbe rimosso e la memoria aggiornata.

Sono possibili molte variazioni di questo semplice protocollo. Per esempio, una snooping cache potrebbe evitare di invalidare un proprio elemento a seguito di un write hit, ma anzi, potrebbe aggiornarlo con il nuovo valore. Dal punto di vista concettuale questo comportamento è equivalente a un'invalidazione seguita da un caricamento da memoria principale. In tutti i protocolli di cache bisogna scegliere se adottare una **strategia di aggiornamento** o una **d'invalidazione**.

La scelta produce protocolli che esibiscono prestazioni differenti a seconda dei carichi di lavoro. I messaggi di aggiornamento contengono le informazioni da adeguare e sono perciò più grandi di quelli d'invalidazione, ma prevengono miss futuri della cache.

Un'altra variante prevede di effettuare un caricamento nella snooping cache anche in caso di write miss. La correttezza dell'algoritmo non è compromessa dal caricamento, ma le prestazioni lo sono. La domanda è: "Qual è la probabilità che una parola appena scritta venga riscritta presto?" Se è alta, allora ci sono argomenti a favore del caricamento nella cache dopo un write miss, e si parla di una **politica write-allocate**. Se invece la probabilità è bassa, è preferibile non aggiornare dopo un write miss: se la parola viene letta poco dopo, verrà caricata comunque a seguito di un read miss, perciò si perde poco a non caricarla dopo un write miss.

Così come gran parte delle soluzioni semplici, anche questa è inefficiente. Ogni operazione di scrittura raggiunge la memoria tramite il bus, che quindi risulterà un collo di bottiglia anche con un numero modesto di CPU. Per contenere l'aumento del traffico sul bus sono stati ideati altri protocolli di cache, che hanno in comune la proprietà che non tutte le scritture raggiungono direttamente la memoria. Quando viene modificata una linea di cache, viene asserito al suo interno un bit che annota la sua validità in contrapposizione alla memoria. La linea così modificata dovrà essere scritta prima o poi in memoria, ma ciò si potrebbe verificare dopo molte scritture al suo interno. Questi tipi di protocollo si chiamano **protocolli write-back.**

Protocollo MESI di coerenza delle cache

MESI è un protocollo di coerenza delle cache molto diffuso e di tipo write-back (Papamarcos e Patel, 1984). Si chiama così perché si avvale degli stati *Modified, Exclusive, Shared* e *Invalid*, ed è basato su un precedente protocollo detto **write-once** (Goodman, 1983). Il protocollo MESI è utilizzato dal Pentium 4 e da molte altre CPU per lo snooping sul bus. I quattro stati in cui si possono trovare gli elementi della cache sono i seguenti.

1. Non valido: l'elemento di cache non contiene dati validi.
2. Condiviso: la linea potrebbe essere contenuta in più di una cache; la memoria è aggiornata.
3. Esclusivo: nessun'altra cache contiene la linea; la memoria è aggiornata.
4. Modificato: l'elemento è valido; la memoria non lo è; non ci sono altre copie della linea.

Quando una CPU viene avviata, tutti gli elementi di cache sono contrassegnati come non validi. Alla prima lettura la linea referenziata viene prelevata dalla memoria e inserita nella cache della CPU; il suo stato è esclusivo, perché è la sola copia presente in una cache, come illustrato nella Figura 8.26(a) con riferimento alla CPU 1 dopo la lettura della linea *A*. Le letture successive a opera di quella CPU usano l'elemento della cache e non transitano sul bus. È possibile che un'altra CPU prelevi la stessa linea di cache, ma grazie allo snooping la CPU 1 sa di non essere più la sola detentrice e annuncia sul bus di possedere una copia della linea. Entrambe le copie vengono contrassegnate come S (*Shared*), come mostrato nella Figura 8.26(b). Detto altrimenti, lo stato S significa che la linea si trova in una o più cache per la lettura e che la memoria è aggiornata. Le letture di una CPU all'interno di una linea nello stato S non si avvalgono del bus e non ne modificano lo stato.

Si consideri ora che cosa succede se la CPU 2 scrive in una sua linea di cache che si trova nello stato S. La CPU emette un segnale d'invalidazione sul bus, indicando alle altre CPU di eliminare le proprie copie. La copia passa ora allo stato M, come mostrato nella Figura 8.26(c). La linea non viene scritta in memoria. Val la pena notare che se una linea si trova nello stato E al momento di una scrittura, non ci vuole alcun segnale d'invalidazione a beneficio delle altre cache, perché è chiaro che non esistono altre copie.

Si consideri poi che cosa succede quando la CPU 3 legge la linea. La CPU 2 detiene ora la linea e sa che la copia in memoria non è valida, perciò invia sul bus un segnale che indica alla CPU 3 di attendere mentre la linea viene scritta in memoria. Al completamento

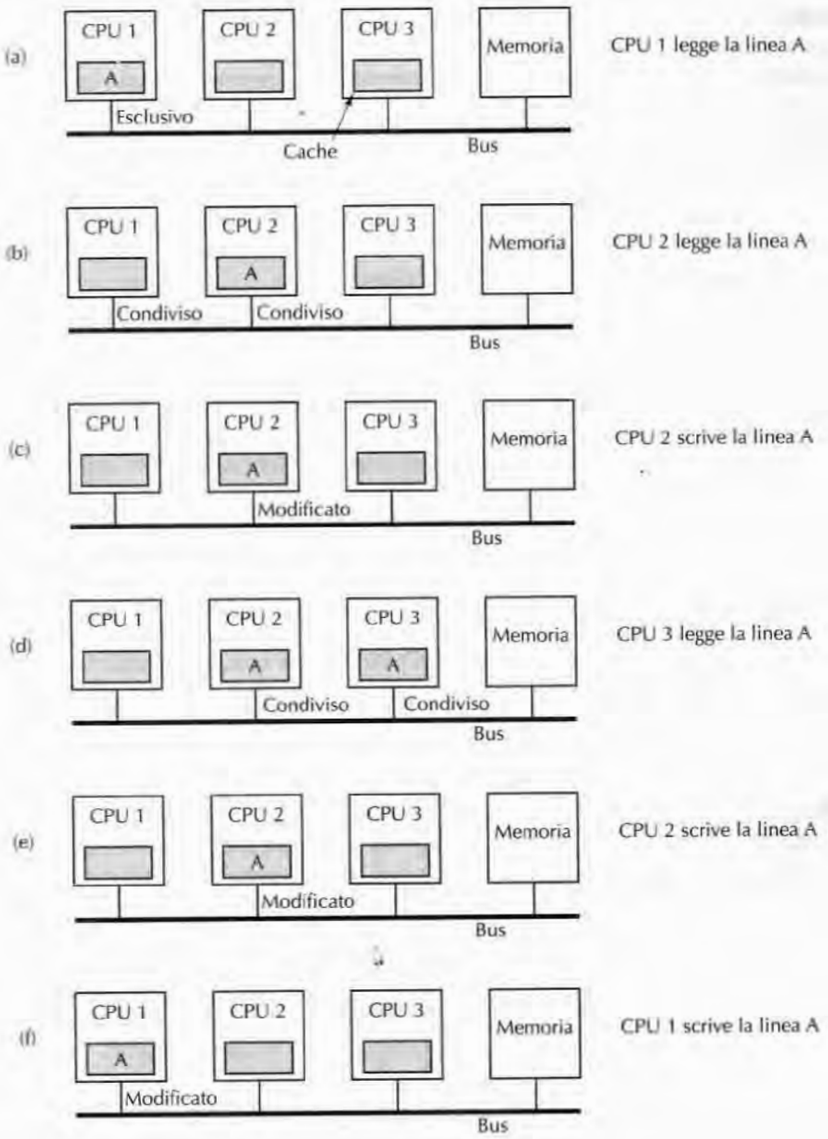


Figura 8.26 Protocollo MESI di coerenza delle cache.

dell'operazione, la CPU 3 preleva la sua copia e la linea viene contrassegnata come condivisa da entrambe le cache, come mostrato dalla Figura 8.26(d). Dopo di ciò, la CPU 2 scrive ancora nella linea, il che invalida la copia della CPU 3 (Figura 8.26(e)).

Infine anche la CPU 1 scrive una parola nella linea. La CPU 2 osserva il tentativo di scrittura e invia un segnale sul bus che richiede alla CPU 1 di attendere la scrittura della linea in memoria. Al completamento dell'operazione contrassegna la propria linea come non valida, perché sa che un'altra CPU sta per modificarla. A questo punto ci possiamo trovare nella situazione in cui una CPU sta scrivendo all'interno di una linea fuori dalla cache. Se si usa la politica write-allocate, la linea viene caricata nella cache e contrassegnata come modificata, come succede nella Figura 8.26(f). Altrimenti, la scrittura avviene direttamente in memoria e la linea non viene caricata in alcuna cache.

Multiprocessori UMA con commutatori crossbar

Per quanto se ne possa ottimizzare l'uso, la presenza di un solo bus limita il parallelismo dei multiprocessori UMA a circa 16 o 32 CPU. Per superare questo valore ci vuole un diverso tipo di rete d'interconnessione. Il circuito più semplice che collega n CPU a k memorie è il **commutatore crossbar** ("a traversa") mostrato nella Figura 8.27. I commutatori crossbar sono stati usati per decenni all'interno della cabine di commutazione telefoniche per poter collegare in qualsiasi modo un certo numero di linee in ingresso a un insieme di linee in uscita.

A ogni intersezione tra una linea orizzontale (in ingresso) e una verticale (in uscita) c'è un **crosspoint** ("punto di incrocio"). Un crosspoint è esso stesso un piccolo commutatore che può essere aperto o chiuso elettronicamente, a seconda che si voglia collegare o meno le linee corrispondenti. La Figura 8.27(a) mostra tre crosspoint chiusi contemporaneamente,

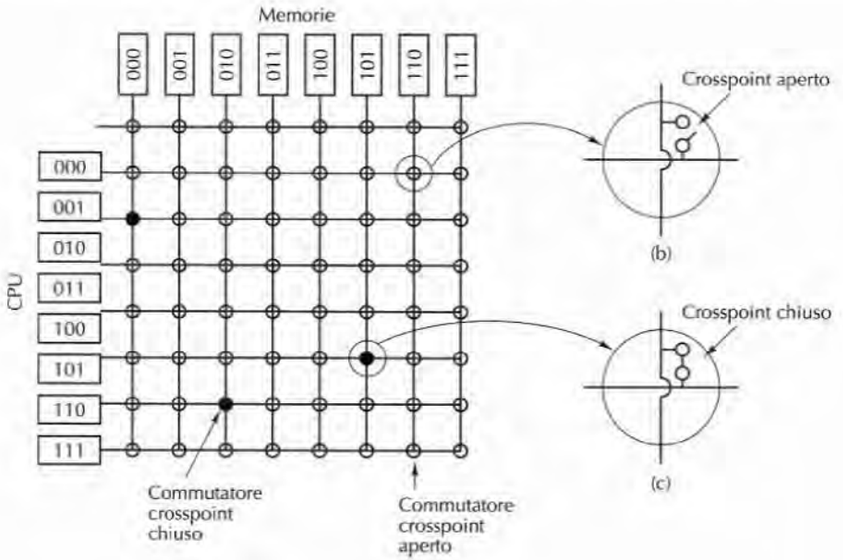


Figura 8.27 (a) Commutatore crossbar 8 × 8. (b) Crosspoint aperto. (c) Crosspoint chiuso.

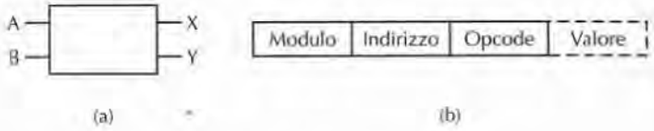


Figura 8.28 (a) Commutatore 2 × 2. (b) Formato di messaggio.

consentendo così la comunicazione simultanea tra le seguenti coppie (CPU, memoria): (001, 000), (101, 101) e (110, 010). Sono possibili molte altre combinazioni; infatti il numero di combinazioni è uguale ai diversi modi in cui si possono sistemare otto torri su di una scacchiera senza che si attacchino[4].

Una delle proprietà più apprezzabili dei commutatori crossbar è che costituiscono **reti non bloccanti**, ovvero non succede mai che venga negata a una CPU la connessione di cui ha bisogno perché la linea è già occupata (ipotizzando che il modulo di memoria desiderato sia disponibile). Inoltre non c'è bisogno di alcuna pianificazione: anche se sono già state stabilite sette connessioni qualsiasi, è sempre possibile collegare un'ottava CPU a una nuova memoria. Vedremo più avanti alcuni schemi d'interconnessione che non hanno queste proprietà.

Uno dei difetti di questi crossbar è che il numero di crosspoint cresce come n^2. Un progetto può pensare di adottare un commutatore crossbar se il sistema in questione è di medie dimensioni. Più avanti nel capitolo incontreremo un progetto di questo tipo: il Sun Fire E25K. D'altra parte, se ci sono 1000 CPU e 1000 moduli di memoria ci vorrebbero un milione di crosspoint, il che non è fattibile in pratica. In una situazione del genere c'è bisogno di qualcosa di molto diverso.

Multiprocessori UMA con reti a commutazione multilivello

Quel "qualcosa di molto diverso" può essere l'umile commutatore 2 × 2 della Figura 8.28(a). Questo commutatore ha due ingressi e due uscite; i messaggi che provengono da ciascuno dei due input possono essere indirizzati verso ognuna delle due uscite. Ipotizziamo per comodità che i messaggi siano composti di quattro parti, come schematizzato nella Figura 8.28(b). *Modulo* specifica la memoria da usare. *Indirizzo* specifica un indirizzo all'interno di quel modulo. *Opcode* stabilisce il tipo di operazione, per esempio `READ` o `WRITE`. Infine il campo opzionale *Valore* potrebbe contenere un operando, come una parola di 32 bit da scrivere con una `WRITE`. Il commutatore esamina il campo *Modulo* e lo usa per determinare la linea X o Y lungo cui inviare il massaggio.

I nostri commutatori 2 × 2 possono essere composti in molti modi per dar vita a una più grande **rete a commutazione multilivello**. Una possibilità economica e senza fronzoli è la **rete omega** (illustrata nella Figura 8.29). In questo caso ci sono otto CPU connesse a otto memorie tramite 12 commutatori. In generale, per collegare n CPU a n memorie ci vogliono $\log_2 n$ livelli di $n/2$ commutatori ciascuno, per un totale di $(n/2) \log_2 n$ commutatori: è un numero molto inferiore a n^2, specie per valori grandi di n.

---

[4] Si tratta del famoso problema delle Non-Attacking Rooks (*N.d.Rev.*).

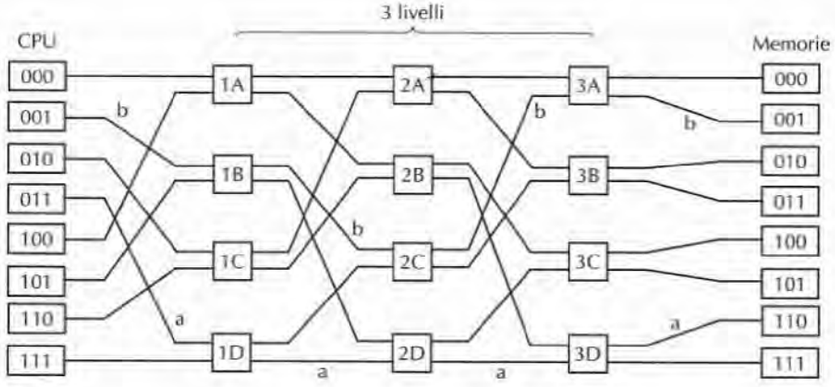


Figura 8.29 Rete di connessione omega.

Spesso ci si riferisce alla struttura dei collegamenti di queste reti con il termine di **mescolamento perfetto** (*perfect shuffle*) perché il mescolamento dei segnali a ogni livello somiglia a un mazzo di carte tagliato in due e rimescolato carta per carta. Per capire il funzionamento di una rete omega, supponiamo che la CPU 011 voglia leggere una parola dal modulo di memoria 110. La CPU spedisce un messaggio READ contenente 110 nel campo *Modulo* al commutatore 1D. Il commutatore preleva il primo bit (quello più a sinistra) da 110 e lo usa per l'instradamento. Lo 0 instrada verso l'uscita in alto, 1 verso la linea in basso. Poiché il bit vale 1 il messaggio viene instradato a 2D attraverso l'uscita in basso.

Tutti i commutatori di secondo livello, compreso 2D, usano per l'instradamento il secondo bit. In questo caso è ancora un 1 e così il messaggio viene spedito a 3D attraverso l'uscita in basso. Qui viene testato il terzo bit che vale 0. Di conseguenza il messaggio fuoriesce dalla linea in alto e raggiunge la memoria 110, come richiesto. Il cammino percorso da questo messaggio è indicato con una lettera *a* nella Figura 8.29.

Mentre i messaggi attraversano la rete di commutazione, i bit più a sinistra del campo *Modulo* diventano progressivamente inutili. Possono perciò essere riutilizzati registrando al loro interno i numeri delle linee in ingresso, in modo da specificare il percorso di ritorno del messaggio di risposta. Nel caso del cammino *a* le linee in ingresso sono 0 (ingresso in alto di 1D), 1 (ingresso in basso di 2D) e 1 (ingresso in basso di 3D). La risposta viene instradata usando 011, che questa volta va letto da destra a sinistra.

Mentre si svolgono queste operazioni, la CPU 001 vuole scrivere una parola nel modulo di memoria 001. Il procedimento è analogo al precedente: il messaggio viene instradato rispettivamente attraverso le uscite in alto, in alto e poi in basso, come indicato dalla lettera *b*. Alla consegna del messaggio *Modulo* contiene 001, che rappresenta il cammino percorso. Le due richieste possono procedere parallelamente dal momento che usano commutatori, linee e memorie disgiunti.

Si consideri ora che cosa accadrebbe se la CPU 000 volesse accedere al modulo di memoria 000. La sua richiesta entrerebbe in conflitto con quella della CPU 001 in corrispondenza dello switch 3A. Una delle due dovrebbe aspettare; a differenza del commutatore crossbar la rete omega è una **rete bloccante**. Non è possibile elaborare simultaneamente qualsiasi insieme di richieste e i conflitti possono verificarsi per l'uso di un collegamento o di un commutatore sia tra richieste che *vanno* alla memoria, sia tra quelle che *provengono* da essa.

È quindi auspicabile riuscire a distribuire i riferimenti alla memoria in modo uniforme rispetto ai moduli. Una tecnica comune è quella di usare i bit meno significativi come numero di modulo. Si consideri per esempio uno spazio degli indirizzi orientato al byte in un calcolatore che accede prevalentemente a parole di 32 bit. I due bit meno significativi si troveranno spesso a 00, ma i 3 bit successivi saranno distribuiti uniformemente. Se si usano questi 3 bit come numero di modulo, le parole indirizzate in modo consecutivo si troveranno in moduli consecutivi. Un sistema di memoria in cui le parole consecutive si trovano in moduli diversi si dice **interlacciato**. Le memorie interlacciate massimizzano il parallelismo perché la maggior parte dei riferimenti a memoria coinvolge indirizzi consecutivi. È altresì possibile progettare reti di commutazione non bloccanti e che offrono cammini multipli da ogni CPU a ogni modulo di memoria, per meglio distribuire il traffico di rete.

8.3.4 Multiprocessori NUMA

Dovrebbe essere ora chiaro che i multiprocessori UMA a bus singolo sono spesso limitati a non più di qualche dozzina di CPU, e i multiprocessori con commutazione crossbar o multilivello necessitano di molto hardware (costoso) e perciò non possono essere molto più grandi. Per superare le 100 CPU bisogna rinunciare a qualcosa, spesso all'idea che tutti i moduli di memoria richiedano lo stesso tempo d'accesso. Questa rinuncia conduce ai multiprocessori **NUMA** (*NonUniform Memory Access*) che, come i cugini UMA, gestiscono un solo spazio degli indirizzi per tutte le CPU, ma diversamente da loro, garantiscono un accesso più veloce ai moduli vicini rispetto a quelli lontani. Dunque tutti i programmi UMA continuano a funzionare sulle macchine NUMA, ma le prestazioni degradano molto rispetto alle macchine UMA che hanno la stessa frequenza di clock.

Le macchine NUMA hanno sempre tre caratteristiche che le accomunano e le distinguono dagli altri multiprocessori.

1. C'è un solo spazio degli indirizzi visibile a tutte le CPU.
2. L'accesso alla memoria distante si effettua tramite istruzioni LOAD e STORE.
3. L'accesso alla memoria distante è più lento di quello alla memoria locale.

Quando il tempo di accesso alla memoria distante non è nascosto (perché non c'è caching) si parla di sistemi **NC-NUMA**. Viceversa, quando sono presenti cache coerenti si parla di **CC-NUMA** (almeno tra gli esperti di hardware). Gli esperti di software spesso usano il termine **DSM hardware** perché si tratta fondamentalmente di una memoria condivisa distribuita, ma questa volta implementata in hardware con una dimensione di pagina piccola.

Una macchina NC-NUMA ante litteram fu il Carnegie-Mellon Cm*, illustrato in forma semplificata nella Figura 8.30 (Swan et al., 1977). Cm* conteneva varie CPU LSI-11, ciascuna dotata di una piccola memoria cui accedeva tramite un bus locale (il processore LSI-11 era una versione a singolo chip del DEC PDP-11, un minicomputer molto diffuso negli anni '70). Inoltre, i chip LSI-11 erano collegati attraverso un bus di sistema. Quando la MMU (opportunamente modificata) riceveva una richiesta di accesso a memoria, stabiliva per prima cosa se la parola si trovava in memoria locale. In tal caso la richiesta veniva inoltrata lungo il bus locale per ottenere la parola. In caso contrario, la richiesta veniva instradata lungo il

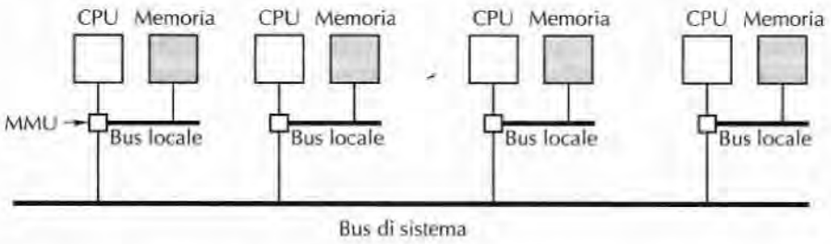


Figura 8.30 Macchina NUMA con due livelli di bus. Il Cm* fu il primo multiprocessore ad adottare questo schema.

bus di sistema verso il processore che conteneva la parola, che rispondeva di conseguenza. Naturalmente questa seconda evenienza impiegava molto più tempo della prima. Anche se un programma poteva girare senza problemi usando solo la memoria distante, impiegava una tempo circa 10 volte maggiore rispetto alla sua esecuzione basata sulla sola memoria locale.

La coerenza della memoria in una macchina NC-NUMA è garantita dal fatto che non c'è caching. Ogni parola di memoria si può trovare in una sola locazione, perciò non c'è pericolo che ne esista una copia difforme: non ci sono copie. D'altro canto, la collocazione di una pagina in memoria diventa molto importante perché il degrado delle prestazioni dovuto a un cattivo posizionamento dei dati è molto rilevante. In ragione di ciò, le macchine NC-NUMA usano software molto elaborato per spostare le pagine all'interno del sistema, in modo da garantire le massime prestazioni.

Generalmente esiste un processo demone chiamato **scanner delle pagine**, eseguito a intervalli di pochi secondi. Il suo compito è esaminare le statistiche di utilizzo e spostare le pagine nel tentativo d'incrementare le prestazioni. Se una pagina sembra posizionata male, lo scanner delle pagine la rimuove dalla mappa delle pagine e così l'accesso successivo a essa causerà un errore di pagina. L'errore di pagina è l'occasione per decidere dove "collocare" la pagina, probabilmente in una memoria diversa da quella in cui si trovava prima. Per evitare il *thrashing* si stabilisce la regola secondo cui, una volta caricata, una pagina viene lasciata al proprio posto per almeno un dato numero di secondi. Sono stati studiati molti algoritmi per la paginazione nelle macchine NC-NUMA, ma nessuno di loro si comporta al meglio in ogni circostanza (LaRowe e Ellis, 1991).

Multiprocessori NUMA con cache coerente

I progetti di multiprocessori come quello della Figura 8.30 risultano insoddisfacenti al crescere delle dimensioni dei sistemi, perché non prevedono il caching. Dover accedere alla memoria remota ogni volta che si fa riferimento a una parola di memoria non locale vuol dire pagare un alto prezzo in termini di prestazioni. Se però si aggiunge il caching, bisogna assicurare anche la coerenza delle cache. Un modo possibile per garantirla è permettere lo snooping del bus di sistema. Si tratta di una soluzione facile da implementare dal punto di vista tecnico, ma che ben presto si rivela praticamente inattuabile al crescere del numero di CPU. La costruzione di multiprocessori veramente grandi richiede un approccio completamente diverso.

Attualmente il modo più diffuso per costruire grandi **CC-NUMA** (*Cache Coherent NUMA Multiprocessor*, "multiprocessori NUMA con coerenza della cache") è il sistema **multiprocessore basato su directory**. L'idea è quella di mantenere un database per ricordare la collocazione e lo stato di ciascuna linea di cache. Dopo un riferimento a una linea di cache, si interroga il database per scoprire dove si trova la linea e se è intatta o è stata modificata. Poiché il database viene interrogato a ogni istruzione che accede alla memoria, è necessario che sia implementato con hardware specializzato molto veloce, capace di rispondere in una frazione di ciclo di bus.

Per meglio circostanziare l'idea alla base di questi multiprocessori, consideriamo un semplice esempio di un sistema di 256 nodi, ciascuno costituito da una CPU e da una RAM di 16 MB, collegate tramite un bus locale. La memoria totale è di 2^{32} byte, divisi in 2^{26} linee di cache di 64 byte ciascuna. La memoria è allocata staticamente ai diversi nodi, laddove i primi 16 MB sono attribuiti al nodo 0, le locazioni da 16 a 32 MB al nodo 1, e così via. Si dice che il nodo 0 è il nodo di appartenenza (*home node*) dei blocchi di memoria dei primi 16 MB e delle corrispondenti linee di cache. I nodi sono collegati per mezzo della rete d'interconnessione rappresentata nella Figura 8.31(a). La rete d'interconnessione potrebbe essere una griglia, un ipercubo o avere un'altra topologia. Ogni nodo contiene anche gli elementi di directory per le 2^{18} linee di cache da 64 byte che costituiscono i suoi 2^{24} byte di memoria. Ipotizziamo per il momento che ogni linea si possa trovare al massimo in una cache.

Per comprendere il funzionamento della directory seguiamo un'istruzione `LOAD` della CPU 20, che fa riferimento a una certa linea di cache. All'inizio la CPU emette l'istruzione e la passa alla propria MMU, che la traduce in un indirizzo fisico, diciamo 0x24000108. La MMU divide l'indirizzo nelle tre parti mostrate nella Figura 8.31(b). In decimale, le tre parti

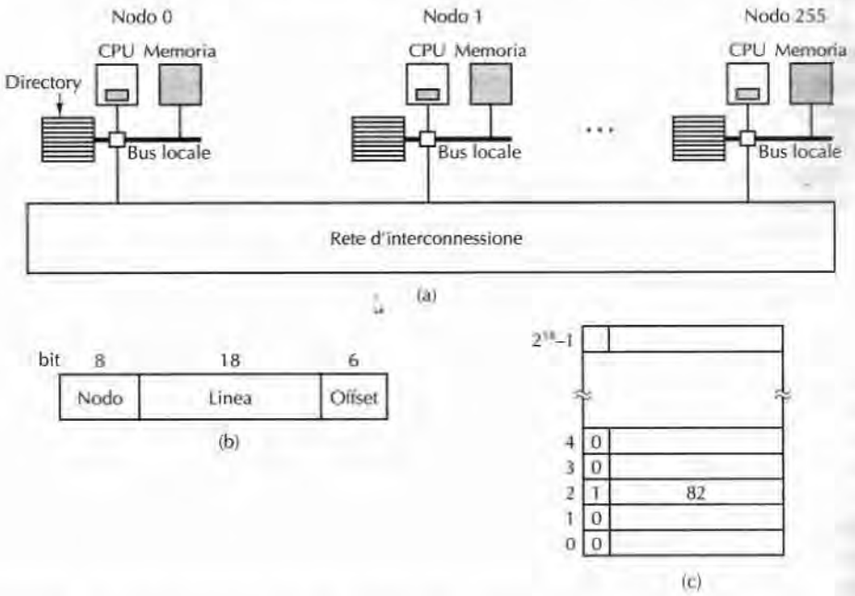


Figura 8.31 (a) Multiprocessore di 256 nodi basato su directory. (b) Divisione in campi di un indirizzo di memoria di 32 bit. (c) Directory in corrispondenza del nodo 36.

valgono: nodo 36, linea 4, offset 8. La MMU si accorge che il riferimento è a una parola di memoria del nodo 36, non del nodo 20, perciò invia un messaggio di richiesta attraverso la rete d'interconnessione verso il nodo che ospita la linea, cioè il 36, e gli richiede se abbia in cache la linea 4 e, in caso affermativo, dove si trovi.

La richiesta, pervenuta al nodo 36 dalla rete d'interconnessione, viene instradata all'hardware della directory, che la usa come indice per la sua tabella di 2^{18} elementi (uno per ogni linea di cache) per estrarre l'elemento 4. Nella Figura 8.31(c) osserviamo che la linea non è presente in cache, perciò l'hardware preleva la linea 4 dalla RAM locale, la spedisce al nodo 20 e aggiorna l'elemento 4 della directory a indicare che la linea è adesso presente nella cache del nodo 20.

Prendiamo ora in considerazione una seconda richiesta, questa volta riguardante la linea 2 del nodo 36. La Figura 8.31(c) evidenzia che la linea è presente nella cache presso il nodo 82. A questo punto l'hardware potrebbe aggiornare l'elemento 2 della directory per specificare che la linea si trova ora nel nodo 20 e inviare quindi un messaggio al nodo 82 per ordinargli di consegnare la linea al nodo 20 e di invalidare la propria cache. Si noti che anche un cosiddetto "multiprocessore a memoria condivisa" nasconde una certa mole di scambi di messaggi.

Prendiamoci un attimo per calcolare lo spazio di memoria richiesto dalle directory. Ogni nodo ha 16 MB di RAM e 2^{18} elementi di 9 bit per mantenere traccia della RAM. Perciò l'informazione accessoria richiesta dalla directory è di circa 9×2^{18} bit diviso 16 MB, cioè circa l'1,76%; è una quantità abbastanza accettabile (anche se si tratta di memoria ad alta velocità, il che ne incrementa i costi). Se le linee di cache fossero di 32 byte, la percentuale salirebbe soltanto al 4%. Con linee di cache di 128 byte sarebbe invece al di sotto dell'1%.

La restrizione evidente imposta da questo progetto è che una linea si trovi per lo più nella cache di un solo nodo. Per permettere il caching delle linee in più nodi si dovrebbe trovare un modo per localizzarle tutte, in modo da poterle invalidare o aggiornare dopo una scrittura. Per permettere il caching simultaneo all'interno di più nodi sono possibili diverse opzioni.

Una possibilità è dotare ogni elemento di directory di k campi atti a individuare altri nodi, consentendo così il caching di una linea in un massimo di k nodi. Una seconda possibilità è la sostituzione del numero di nodo del nostro semplice progetto con una bit map, in cui a ogni bit corrisponde un nodo. Secondo questa opzione non c'è alcun limite al numero di copie di una linea, ma c'è un sostanziale incremento del volume d'informazioni accessorie. Una directory con 256 bit per ogni linea di cache di 64 byte (512 bit) comporta una percentuale d'informazioni accessorie superiore al 50%. Una terza possibilità è di gestire un campo da 8 bit in ogni elemento di directory e usarlo come puntatore a una lista concatenata contenente tutte le copie della linea di cache corrispondente. Questa strategia richiede sia spazio accessorio per i puntatori alle liste, sia tempo di esecuzione per scorrere le liste quando è necessario individuare tutte le copie di una linea. Ciascuna possibilità presenta vantaggi e svantaggi, e tutte e tre sono state usate su sistemi reali.

Un altro miglioramento che si può apportare al progetto di directory è la capacità di ricordare lo stato di un linea: "pulita" o "sporca", a seconda che la memoria di appartenenza sia o meno aggiornata. Se giunge una richiesta per una linea di cache pulita, il nodo di appartenenza può soddisfare la richiesta direttamente dalla memoria, senza dover accedere alla cache. Tuttavia, una richiesta di lettura di una linea di cache sporca deve essere inoltrata al nodo contenente la linea di cache, perché è il solo ad averne una copia valida. Se è consentita una sola copia per ogni linea di cache, come nella Figura 8.31, non si ottiene alcun miglioramento reale a mantenere traccia dello stato delle linee, perché ogni nuova richiesta comporta l'invio di un messaggio alla copia esistente per invalidarla.

Naturalmente se si tiene traccia dello stato di una linea bisogna anche informare il suo nodo di appartenenza ogni volta che viene modificata, anche se ne esiste una sola copia. Se invece esistono molteplici copie, la modifica di una di loro richiede l'invalidazione di tutte le altre, perciò si rende necessario un protocollo per evitare le corse critiche. Per esempio, per modificare una linea di cache condivisa, uno dei suoi detentori potrebbe dover richiedere l'accesso esclusivo *prima* della modifica. Una richiesta di questo tipo causerebbe l'invalidazione di tutte le altre copie prima di permettere l'accesso. Altre ottimizzazioni per le prestazioni delle macchine CC-NUMA sono trattate in (Stenstrom et al., 1997).

Il multiprocessore NUMA Sun Fire E25K

Studiamo ora la famiglia di processori Sun Fire di Sun Microsystems come esempio di multiprocessori NUMA a memoria condivisa. Benché questa famiglia raggruppi vari modelli, focalizzeremo la nostra attenzione sul modello E25K, equipaggiato con 72 CPU UltraSPARC IV. Un UltraSPARC IV è dotato di due processori UltraSPARC III che condividono una cache e una memoria. Il modello E15K è del tutto analogo all'E25K, se non che è costruito a partire da processori singoli invece che da chip a due processori. Esistono anche modelli più piccoli di questi, ma dal nostro punto di vista ci interessiamo proprio al funzionamento di quelli contenenti il maggior numero di CPU.

Il sistema E25K è formato da 18 insiemi di schede: ogni insieme comprende una scheda di CPU e memoria, una scheda di I/O con quattro alloggiamenti PCI e una scheda di espansione che serve a collegare le due schede precedenti al piano centrale, che sorregge tutte le schede e contiene i circuiti per la commutazione. Ogni scheda di CPU e memoria contiene quattro chip di CPU e quattro moduli di RAM da 8 GB. Dunque ogni scheda di CPU e memoria dell'E25K contiene otto CPU (quattro coppie) e 32 GB di RAM (quattro CPU e 32 GB di RAM nell'E15K). Un E25K completo (illustrato nella Figura 8.32) contiene perciò 144 CPU, 576 GB di RAM, 72 alloggiamenti PCI. È interessante sapere che il numero 18 è frutto di vincoli di imballaggio: il sistema con 18 insiemi di schede si è rivelato il più grande sistema che potesse attraversare i vani delle porte senza essere smontato. Laddove i programmatori pensano solo in funzione di 0 e 1, gli ingegneri devono preoccuparsi anche di altro, per esempio del fatto che i clienti riescano a trasportare il loro prodotto all'interno degli edifici passando attraverso le porte.

Il piano centrale si compone di un insieme di tre commutatori crossbar 18×18 per la connessione dei 18 gruppi di schede. Un crossbar serve per le linee di indirizzi, uno per le risposte e uno per il trasferimento dati. Oltre alle 18 schede di espansione, il piano centrale è collegato anche a un insieme di schede di controllo del sistema contenente una sola CPU, ma che ha interfacce per CD-ROM, periferiche a nastro, linee seriali e altri dispositivi di periferica necessari al riavvio, alla manutenzione e al controllo del sistema.

Il cuore di ogni multiprocessore è il sottosistema di memoria. Come si possono connettere 144 CPU alla memoria distribuita? I modi diretti, un grosso snooping bus condiviso o un commutatore crossbar 144×72, non funzionano bene. Il primo non conviene perché il bus diventa un collo di bottiglia, il secondo perché il commutatore risulta troppo difficile e costoso da costruire. Per queste ragioni i multiprocessori come l'E25K sono costretti a usare un sottosistema di memoria più complesso.

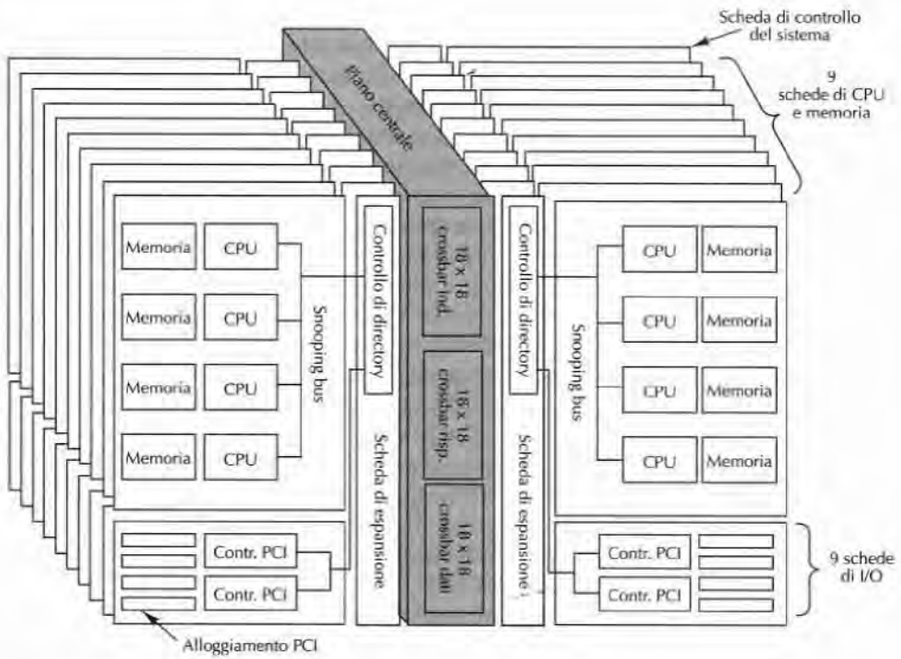


Figura 8.32 Multiprocessore E25K di Sun Microsystems.

A livello degli insiemi di schede si utilizza lo snooping di modo che tutte le CPU locali possano esaminare tutte le richieste di memoria che provengono dallo stesso insieme di schede e che referenziano i blocchi che si trovano nelle loro cache. Perciò quando una CPU necessita di una parola di memoria, per prima cosa converte l'indirizzo virtuale in un indirizzo fisico e controlla la propria cache (gli indirizzi fisici sono di 43 bit, ma per limitare il volume del multiprocessore la memoria è di soli 576 GB). Se la linea di cache richiesta si trova nella sua cache, la parola viene prelevata direttamente. Altrimenti, i circuiti di snooping verificano se esiste una copia di quella parola in qualche altra locazione dell'insieme di schede. Se così fosse, la richiesta verrebbe soddisfatta. Se invece la ricerca è infruttuosa, la richiesta viene trasferita sul commutatore crossbar 18 × 18 degli indirizzi, come descritto in seguito. La logica di snooping può effettuare un controllo per ogni ciclo di clock. Visto che il clock oscilla a 150 MHz, è possibile effettuare 150 milioni di operazioni di snooping al secondo all'interno di ciascun insieme di schede, ovvero 2,7 miliardi di snooping/s nell'intero sistema.

Anche se da un punto di vista logico la circuiteria di snooping è un bus (Figura 8.32) dal punto di vista fisico si tratta di un dispositivo ad albero, in cui i comandi attraversano l'albero dall'alto verso il basso o viceversa. Quando una CPU o una scheda PCI emettono un indirizzo, questo giunge a un ripetitore di indirizzi tramite una connessione diretta (Figura 8.33). I due ripetitori di indirizzi convergono sulla scheda di espansione, da cui gli indirizzi vengono rispediti indietro verso la parte bassa dell'albero, affinché ogni dispositivo possa verificare la presenza della parola richiesta. Questo stratagemma serve a evitare l'implementazione di un bus che colleghi tre schede diverse.

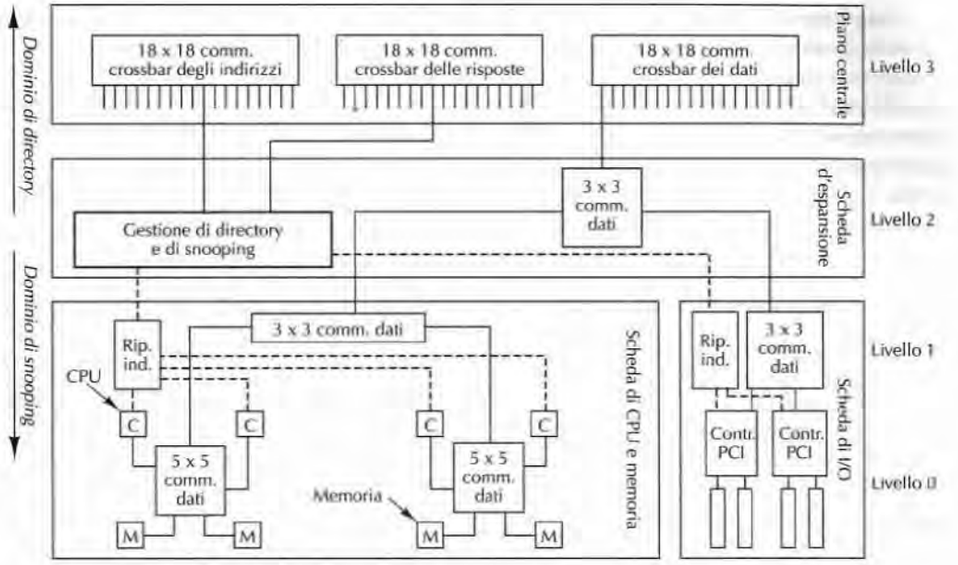


Figura 8.33 Sun Fire E25K usa un'interconnessione a quattro livelli. Le linee tratteggiate rappresentano i percorsi degli indirizzi, quelle continue i percorsi dei dati.

I trasferimenti di dati usano un'interconnessione a quattro livelli (Figura 8.33) allo scopo di raggiungere prestazioni elevate. A livello 0 i chip con le coppie di CPU e le memorie sono collegati tramite un piccolo commutatore crossbar da cui si diparte anche un collegamento verso il livello 1. I due gruppi di memorie e coppie di CPU sono collegati da un secondo commutatore crossbar del livello 1. I commutatori crossbar sono circuiti progettati su misura e ciascuno di loro può prelevare gli input sia dalle righe sia dalle colonne, anche se non tutte le combinazioni vengono utilizzate (e non tutte sono sensate). Tutta la circuiteria di commutazione che si trova sulle schede è costruita a partire da commutatori crossbar 3 × 3.

Ogni insieme di schede ne contiene tre: la scheda di CPU e memoria, la scheda di I/O e la scheda di espansione, che collega le prime due. Il livello 2 d'interconnessione, che si trova sulla scheda di espansione, è un altro commutatore crossbar 3 × 3 che collega la memoria alle porte di I/O (che sono mappate in memoria come in tutti gli UltraSPARC). Tutti i trasferimenti di dati, in ingresso o in uscita dagli insiemi di schede, che siano diretti alla memoria o a una porta di I/O, passano attraverso il commutatore di livello 2. Infine, i dati che devono essere trasferiti verso una scheda distante o che provengono da una di loro, passano attraverso un commutatore crossbar 18 × 18 che si trova a livello 3.

I trasferimenti di dati sono effettuati a gruppi di 32 byte alla volta, perciò l'unità di trasferimento usuale, 64 byte, impiega due cicli di clock.

Dopo aver dato uno sguardo alla disposizione dei componenti, indirizziamo ora la nostra attenzione sulla modalità operativa della memoria condivisa. A livello più basso, i 576 GB di memoria sono suddivisi in 2^{29} blocchi di 64-byte ciascuno. Questi blocchi sono le unità atomiche del sistema di memoria e a ciascuno di loro è associata una scheda di appartenenza (*home board*) che li ospita preferibilmente quando non sono usati da un'altra parte. La maggior parte dei blocchi si trova per gran parte del tempo nelle rispettive schede di appartenenza. Tuttavia, quando una CPU ha bisogno di un certo blocco di memoria, che si trovi sulla sua scheda o su di una delle rimanenti 17, ne richiede una copia per la propria cache e quindi vi accede all'interno della cache. Anche se ogni chip dell'E25K contiene due CPU, queste condividono una sola cache fisica e perciò ne condividono tutti i blocchi.

I blocchi di memoria e le linee di cache di ogni chip di CPU si possono trovare in uno dei tre stati seguenti:

1. accesso esclusivo (in scrittura)
2. accesso condiviso (in lettura)
3. non valido (cioè vuoto).

Quando una CPU vuole leggere o scrivere una parola di memoria, per prima cosa controlla nella propria cache. Se non vi trova la parola, emette una richiesta locale per l'indirizzo fisico corrispondente che viene propagata solo all'interno dello stesso insieme di schede. Se una delle cache nell'insieme di schede contiene la linea richiesta, la logica di snooping rileva il successo della ricerca e risponde alla richiesta. Se la linea si trova nello stato di accesso esclusivo, viene trasferita al richiedente e la copia originale viene contrassegnata come non valida. Se si trova nello stato di accesso condiviso, la cache non risponde perché se la linea è "pulita" la memoria risponde sempre.

Se la logica di snooping non riesce a trovare la linea di cache oppure se questa è presente e condivisa, una richiesta lungo il piano centrale viene spedita verso la scheda di appartenenza della linea per rintracciare il corrispondente blocco di memoria. Lo stato di ciascun blocco di memoria è conservato nei bit ECC del blocco, perciò la scheda di appartenenza può determinare immediatamente il suo stato. Se il blocco non è condiviso oppure è condiviso da una o più schede remote, la memoria di appartenenza è aggiornata e la richiesta può essere soddisfatta dalla memoria della scheda di appartenenza. In tal caso, la copia della linea di cache viene trasmessa attraverso il commutatore crossbar dei dati nell'arco di due cicli di clock, e nel volgere di qualche tempo giunge alla CPU richiedente.

Se la richiesta era di lettura, viene creato un elemento nella directory della scheda di appartenenza a specificare che un nuovo cliente condivide la linea di cache e che la transazione è stata ultimata. Se invece la richiesta era di scrittura, bisogna spedire un messaggio d'invalidazione a tutte le (eventuali) schede che ne posseggono una copia. Così facendo, la scheda richiedente si trova ad avere l'unica copia in circolazione.

Si consideri ora il caso in cui il blocco richiesto si trovi nello stato di accesso esclusivo presso una scheda differente da quella di appartenenza. Quando la scheda di appartenenza riceve la richiesta, cerca nella directory l'identità della scheda remota e spedisce al richiedente un messaggio in cui specifica la locazione della linea di cache. Il richiedente spedisce quindi una richiesta all'insieme di schede corretto e questo, una volta ricevuto il messaggio, risponde inviando la linea di cache. Se si tratta di una richiesta di lettura, la linea viene contrassegnata come condivisa e viene spedita anche una copia alla scheda di appartenenza. Se è invece di scrittura, il destinatario del messaggio invalida la propria copia, di modo che il richiedente possa avere la sua copia esclusiva.

Ogni scheda ha 2^{29} blocchi di memoria, quindi nel caso peggiore ci vorrebbe una directory di 2^{29} elementi per tener traccia di tutti i blocchi. In realtà la directory è molto più piccola, perciò può capitare che non ci sia abbastanza spazio per alcuni elementi (la ricerca all'interno della directory è svolta in maniera associativa). In tale evenienza, la directory di appartenenza è costretta a emettere una richiesta broadcast a tutte le altre 17 schede per localizzare il blocco. Il commutatore crossbar delle risposte è coinvolto nel protocollo di coerenza e aggiornamento delle directory, perché gestisce gran parte del traffico di risposta al richiedente. La divisione del traffico di protocollo su due bus (indirizzi e risposte), e del traffico dati su di un terzo bus, incrementa il volume di dati trasferiti nel sistema.

Grazie alla distribuzione del carico di lavoro tra numerosi dispositivi che risiedono su schede diverse, il Sun Fire E25K raggiunge delle prestazioni molto elevate. Oltre ai 2,7 miliardi di snooping/s già menzionati, il piano centrale può gestire fino a nove trasferimenti simultanei, tra nove schede mittenti e nove destinatarie. Il crossbar dei dati è largo 32 byte, perciò a ogni ciclo di clock è possibile trasferire 288 byte attraverso il piano centrale. Con una frequenza di clock di 150 MHz, ciò equivale a un picco di larghezza di banda cumulativa di 40 GB/s quando tutti gli accessi avvengono a distanza. Se poi il software è in grado di collocare le pagine in modo tale da garantire una predominanza di accessi locali, la larghezza di banda del sistema può superare notevolmente i 40 GB/s.

Rimandiamo a (Charlesworth, 2002; Charlesworth, 2001) per maggiori informazioni tecniche su Sun Fire.

8.3.5 Multiprocessori COMA

Le macchine NUMA e CC-NUMA presentano lo svantaggio per cui i riferimenti alla memoria distante sono molto più lenti degli accessi alla memoria locale. Le macchine CC-NUMA nascondono in una certa misura questa differenza di prestazioni grazie al caching. Malgrado ciò, se la quantità di dati distanti richiesti eccede di molto la capacità della cache, si verificheranno continui fallimenti di cache e le prestazioni degraderanno sensibilmente.

Dunque ci troviamo da una parte con le macchine UMA che hanno prestazioni eccellenti, ma che sono limitate per dimensioni e abbastanza costose; dall'altra ci sono le macchine NC-NUMA che raggiungono dimensioni maggiori, ma richiedono una collocazione delle pagine manuale o semiautomatica, spesso con risultati altalenanti. Il problema è che è difficile predire quali pagine saranno richieste e dove; inoltre, le pagine costituiscono spesso un'unità troppo grande perché le si possa trasferire da una parte all'altra del sistema. Le macchine CC-NUMA, come il Sun Fire E25K, possono manifestare scarse prestazioni quando molte CPU richiedono dati distanti. Tutto considerato, ciascuno di questi progetti presenta delle gravi limitazioni.

Esiste un altro tipo di multiprocessori che cerca di aggirare questi problemi usando ogni memoria principale di CPU come una cache. Questo progetto si chiama **COMA** (*Cache Only Memory Access*, "accesso alla memoria di tipo cache") e non richiede che ogni pagina abbia una macchina prefissata di appartenenza, come succede nelle macchine NUMA e CC-NUMA. Infatti il concetto di pagina perde di significato: lo spazio degli indirizzi fisici è invece diviso in linee di cache che vengono trasferite all'interno del sistema su richiesta.

Le linee non hanno una macchina che le ospita di preferenza, sono un po' come i nomadi di alcuni paesi del Terzo Mondo: la loro casa è il luogo in cui si trovano in quel momento. Una memoria che si limita ad attrarre le linee quando sono richieste si dice **memoria d'attrazione**, in inglese *attraction memory*, in contrapposizione a *home memory*. L'utilizzo della RAM principale come una grande cache aumenta notevolmente il tasso di hit e dunque le prestazioni.

Sfortunatamente, come al solito, niente si ottiene per niente. I sistemi COMA introducono due nuovi problemi:

1. come localizzare le linee di cache
2. che cosa succede quando viene estromessa dalla memoria l'ultima copia di una linea.

Il primo problema riguarda il fatto che, quando la MMU ha tradotto l'indirizzo virtuale in un indirizzo fisico, se la linea non si trova nell'hardware della vera cache è difficile stabilire se sia veramente in memoria. L'hardware di paginazione non dà alcun aiuto, perché ogni pagina è fatta di molte linee di cache che vagabondano liberamente. Inoltre, se anche si sapesse che la linea non si trova in memoria, ...dove si trova allora? Non c'è modo di chiederlo alla sua macchina di appartenenza, perché non c'è alcuna macchina di appartenenza.

Esistono delle proposte per risolvere il problema della localizzazione. Al fine di stabilire se una linea di cache si trova in memoria, è possibile aggiungere nuovo hardware che tenga traccia di un'etichetta per ogni linea presente nella cache. La MMU potrebbe poi confrontare l'etichetta della linea richiesta con le etichette delle linee di cache in memoria in cerca di una corrispondenza.

In alternativa, si possono mappare le pagine intere senza richiedere però la presenza di tutte le linee di cache. Secondo questa soluzione, l'hardware dovrebbe tenere una bit map per ogni pagina; ciascun bit corrisponderebbe a una linea di cache e ne indicherebbe la presenza o l'assenza. Questo progetto si chiama **COMA semplice** (S-COMA) e stabilisce che, se una linea è presente, deve trovarsi nella posizione corretta all'interno della sua pagina, ma se non è presente, qualsiasi tentativo di utilizzarla causa una trap che consente al software di cercarla e di caricarla.

A questo punto si rende necessario saper reperire linee veramente distanti. Una soluzione è quella di attribuire a ogni pagina una macchina di appartenenza in termini della collocazione del suo elemento di directory, non dei suoi dati. Così è possibile spedire un messaggio alla macchina di appartenenza quanto meno per localizzare la linea di cache. Un altro schema richiede di organizzare la memoria come un albero e percorrerlo verso l'alto finché non viene trovata la linea ricercata.

Il secondo problema elencato precedentemente equivale a non cancellare l'ultima copia di una linea. Come nelle macchine CC-NUMA, anche qui una linea di cache si può trovare simultaneamente presso più nodi. Al verificarsi di un fallimento di cache si rende necessario il caricamento di una linea, il che spesso richiede l'estromissione di un'altra linea. Che cosa succede se si sceglie una linea che è una copia unica? In tal caso bisogna evitare di cancellarla.

Una soluzione è interrogare la directory e verificare se ci sono altre copie. In caso affermativo si può cancellare la linea senza problemi, altrimenti bisogna trasferirla da qualche altra parte. Un'altra soluzione consiste nell'etichettare una copia di ogni linea come copia originale che non può essere mai cancellata. In questo modo si può evitare il controllo nella directory. Tirando le somme, le macchine COMA promettono prestazioni migliori di quelle CC-NUMA, ma ne esistono ancora pochi esemplari e ci vuole maggiore esperienza per poter giudicare. Le prime due macchine COMA sono state il KSR-1 (Burkhardt et al., 1992) e la Data Diffusion Machine (Hagersten et al., 1992). Una macchina più recente è SDAARC (Eschmann et al., 2002).

8.4 Multicomputer a scambio di messaggi

Come indicato nella Figura 8.21, ci sono due tipi di processori paralleli MIMD: i multiprocessori e i multicomputer. Nel paragrafo precedente abbiamo studiato i primi e abbiamo visto che il loro sistema operativo li percepisce come provvisti di una memoria condivisa cui è possibile accedere tramite le ordinarie istruzioni di `LOAD` e `STORE`. Abbiamo illustrato le possibili implementazioni di questa memoria condivisa: snooping bus, commutazione crossbar, reti a commutazione multilivello e altri schemi basati su directory. Al di là di ciò, i programmi scritti per un multiprocessore possono accedere a ogni locazione di memoria senza sapere nulla della topologia interna o dello schema implementativo. Questa illusione è ciò che rende i multiprocessori così attraenti e la ragione per cui costituiscono un modello di programmazione così gradito agli sviluppatori di software.

A ogni modo, anche i multiprocessori presentano alcune limitazioni ed è per questo che i multicomputer sono altrettanto importanti. In primo luogo, le prestazioni dei multiprocessori peggiorano con il crescere delle loro dimensioni. Abbiamo già visto l'enorme quantità di hardware impiegata da Sun nell'E25K per raggiungere i 72 chip con doppia CPU, mentre in seguito studieremo un multicomputer che ha ben 65.536 CPU. Ci vorranno anni prima che qualcuno riesca a costruire un multiprocessore commerciale da 65.536 nodi, mentre per quel tempo saranno già in uso multicomputer da milioni di nodi.

Oltre a ciò, un altro fattore che può influenzare le prestazioni di un multiprocessore è la contesa di memoria: se 100 CPU cercano di leggere e scrivere sempre le stesse variabili, la contesa delle varie memorie, dei bus e delle directory può comportare un grosso degrado delle prestazioni.

La conseguenza di queste considerazioni è che si pone oggi molto interesse nei multicomputer, cioè nei computer paralleli in cui ogni CPU ha la propria memoria privata, non direttamente accessibile dalle altre. I programmi delle CPU di un multicomputer interagiscono per mezzo delle primitive `send` e `receive` per lo scambio esplicito di messaggi, visto che non possono accedere alle rispettive memorie per mezzo d'istruzioni di `LOAD` e `STORE`. Questa differenza cambia completamente il modello di programmazione.

Ogni nodo di un multicomputer contiene una o più CPU, una certa dose di RAM (che si può pensare condivisa solo tra le CPU di quel nodo), un disco e/o altri dispositivi di I/O, più un processore di comunicazione. I processori di comunicazione sono collegati tramite una rete ad alta velocità di uno dei tipi esaminati nel Paragrafo 8.1.2. Esistono diverse topologie, vari schemi di commutazione e algoritmi d'instradamento, ma c'è un aspetto che tutti i multicomputer hanno in comune: quando un programma applicativo esegue una primitiva `send` il processore di comunicazione riceve una notifica e si incarica di trasmettere il blocco di dati dell'utente presso la macchina di destinazione (eventualmente dopo aver chiesto, e ricevuto, il permesso di farlo).

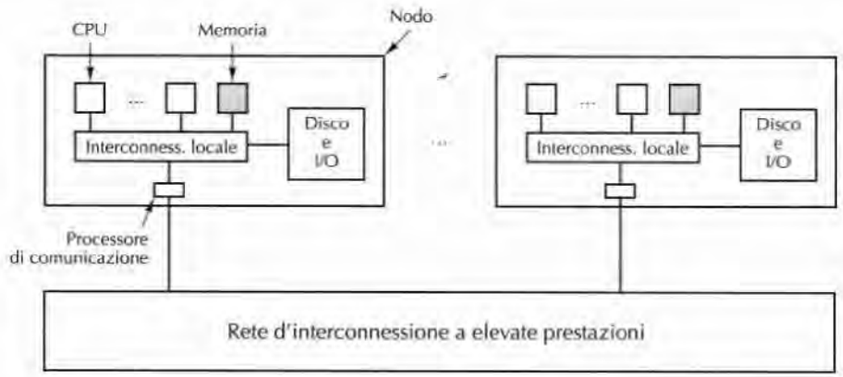


Figura 8.34 Schema di un multicomputer.

8.4.1 Reti d'interconnessione

La Figura 8.34 (che mostra un multicomputer idealizzato) evidenzia il fatto che i multicomputer sono tenuti insieme dalla rete d'interconnessione. Da questo punto di vista i multiprocessori e i multicomputer sono sorprendentemente simili, poiché i multiprocessori sono spesso dotati di moduli di memoria che devono essere interconnessi tra loro e con le CPU. Di conseguenza, quanto trattato in questo paragrafo si applica spesso a entrambi i tipi di sistemi.

La ragione fondamentale della somiglianza tra le reti d'interconnessione dei multiprocessori e quelle dei multicomputer è che in fondo si tratta in entrambi i casi di reti a scambio di messaggi.

Anche nei sistemi monoprocessore, quando la CPU vuole leggere o scrivere una parola, in genere attiva certe linee sul bus e aspetta una risposta. Questa azione è sostanzialmente uno scambio di messaggi: il primo agente invia una richiesta e aspetta una risposta. Nei grandi multiprocessori, la comunicazione tra le CPU e la memoria distante avviene quasi sempre tramite l'invio esplicito di un messaggio di richiesta dati, il cosiddetto **pacchetto**, da parte di una CPU verso la memoria, che replica con un pacchetto di risposta.

Topologia

La topologia di una rete d'interconnessione descrive il modo in cui sono disposti i collegamenti e i commutatori, per esempio lungo un anello o su una griglia. I progetti topologici possono essere modellati con grafi i cui lati sono i collegamenti e i cui nodi sono i commutatori, come nella Figura 8.35. Ogni nodo di una rete d'interconnessione (o del suo grafo) dispone di un certo numero di collegamenti in uscita da esso. I matematici chiamano questo numero il **grado (uscente)** del nodo, mentre gli ingegneri lo chiamano il **fanout** (da *fan out*, "ventaglio di uscite"). In generale, maggiore è il fanout, più numerose sono le scelte d'instradamento e quindi più alta è la tolleranza agli errori: anche se un collegamento è difettoso, la rete può continuare a funzionare se riesce ad aggirarlo. Se ogni nodo ha k lati e se i collegamenti sono effettuati correttamente, è possibile progettare una rete che resti pienamente connessa anche se si interrompono $k - 1$ collegamenti.

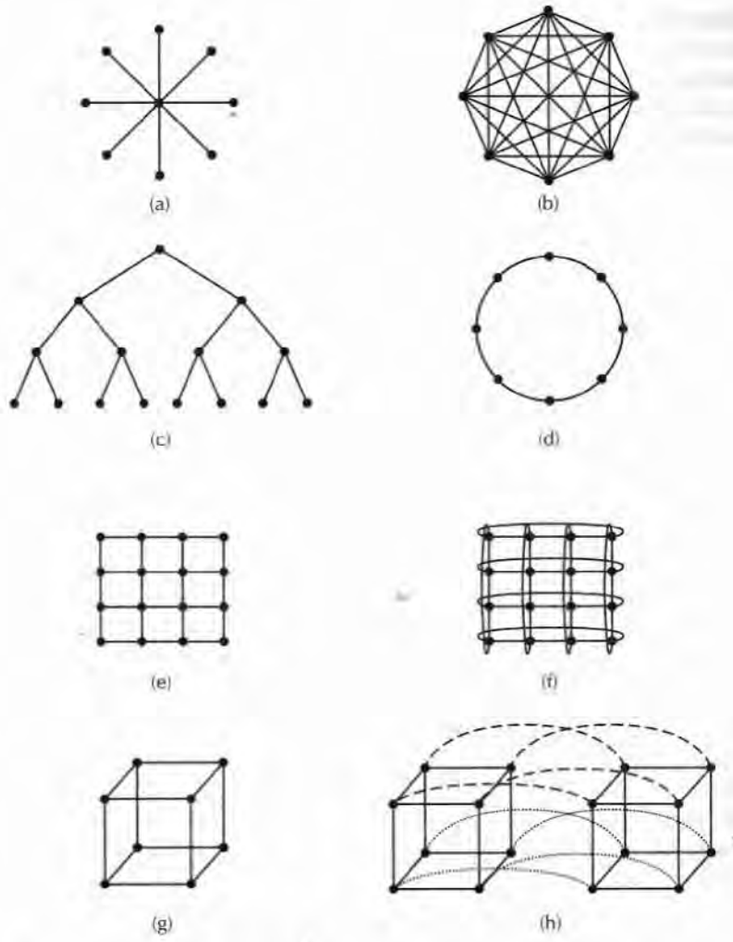


Figura 8.35 Alcune topologie. I pallini rappresentano i commutatori, mentre non sono mostrate le CPU e le memorie. (a) Stella. (b) Interconnessione completa. (c) Albero. (d) Anello. (e) Griglia. (f) Toro. (g) Cubo. (h) Ipercubo a quattro dimensioni.

Un'altra proprietà della rete d'interconnessione (o del suo grafo) è il suo **diametro**. Se misuriamo la distanza tra due nodi con il numero minimo di lati che devono essere attraversati per giungere da uno all'altro, allora il diametro del grafo è la distanza tra due nodi che si trovano alla massima distanza. Il diametro di una rete d'interconnessione è legato al ritardo che si può verificare nel caso peggiore quando si spedisce un pacchetto da una CPU a un'altra o da una CPU a una memoria, dal momento che ogni passaggio attraverso un collegamento impiega una quantità finita di tempo. Più piccolo è il diametro, migliori sono le prestazioni nel caso peggiore. È importante anche la distanza media tra i nodi, poiché influenza il tempo medio di consegna di un pacchetto.

Un'altra proprietà significativa di una rete d'interconnessione è la sua capacità di trasmissione, ovvero la quantità di dati che può trasferire al secondo. Un modo conveniente per misurare questa capacità è usare la **larghezza di banda di bisezione**. Per calcolarla, prima bisogna ripartire (concettualmente) la rete in due parti uguali (per numero di nodi) e scollegarle rimuovendo i lati che collegano nodi di parti diverse. Quindi si calcola la larghezza di banda totale dei lati che sono stati rimossi. Ci possono essere molti modi diversi di bipartire la rete in due parti uguali e la larghezza di banda di bisezione è il valore minimo che si ottiene per tutte le possibili partizioni. Il significato di questo numero è che, se per esempio la larghezza di banda di bisezione è di 800 bit/s, allora, se c'è molta comunicazione in corso tra le due metà, nel caso peggiore la quantità di dati in trasferimento non può superare gli 800 bit/s. Molti progettisti credono che la larghezza di banda di bisezione sia la proprietà più importante per misurare le prestazioni di una rete d'interconnessione, perciò molte reti vengono progettate appositamente per massimizzarla.

Le reti d'interconnessione possono essere caratterizzate dalla loro **dimensionalità**. Per i nostri scopi, la dimensionalità è il numero di modi diversi in cui è possibile raggiungere la destinazione a partire dalla sorgente. Se non c'è mai possibilità di scelta (cioè se c'è solo un cammino possibile da ogni sorgente a ogni destinazione) allora la rete è zero-dimensionale. Se c'è una dimensione in cui è possibile effettuare una scelta, per esempio andare a est o a ovest, allora la rete è uno-dimensionale. Se sono due assi lungo cui si può scegliere (per esempio se un pacchetto può scegliere di proseguire tra est e ovest, oppure tra nord e sud) allora la rete è due-dimensionale, e così via.

La Figura 8.35 mostra alcune topologie e prende in considerazione solo i collegamenti (i segmenti) e i commutatori (i pallini). Le memorie e le CPU non sono mostrate, ma si possono pensare collegate ai commutatori tramite interfacce. Nella Figura 8.35(a) troviamo la configurazione zero-dimensionale a **stella**, in cui le CPU e le memorie sono collegate ai nodi esterni, mentre il nodo centrale è di pura commutazione. Sebbene sia un progetto semplice, la presenza di un nodo centrale commutatore può rivelarsi un collo di bottiglia per sistemi di grandi dimensioni. Inoltre il progetto è scadente anche dal punto di vista della tolleranza agli errori, visto che basta un difetto del commutatore centrale per distruggere l'intero sistema.

La Figura 8.35(b) mostra un altro progetto zero-dimensionale che però si trova all'altro estremo dello spettro: un'**interconnessione completa** (e **completo** è detto il grafo a lei associato). In questo caso ogni nodo è collegato direttamente a ogni altro. Questo progetto massimizza la larghezza di banda di bisezione, minimizza il diametro ed è straordinariamente tollerante agli errori (potrebbero interrompersi sei collegamenti qualsiasi e la rete sarebbe ancora connessa). Sfortunatamente richiede $k(k-1)/2$ lati per collegare k nodi, un numero che diventa presto intrattabile al crescere di k.

Un'altra topologia è l'**albero**, illustrato nella Figura 8.35(c). Il problema di questa topologia è che la larghezza di banda di bisezione è uguale alla capacità dei collegamenti. Dal momento che ci sarà molto traffico in corrispondenza dei nodi vicini alla cima dell'albero, questi pochi nodi diventeranno facilmente un collo di bottiglia per l'intera rete. Una possibile soluzione è aumentare la larghezza di banda di bisezione attribuendo maggiore larghezza di banda ai collegamenti che si trovano più in alto. Per esempio il livello più in basso potrebbe avere capacità b, quello successivo capacità $2b$ e il livello più alto capacità $4b$. Questo tipo di progetto si chiama **fat tree** ("albero grasso") ed è stato utilizzato in alcuni multicomputer commerciali, come l'ormai scomparso CM-5 di Thinking Machines.

L'**anello** della Figura 8.35(d) è, secondo la nostra definizione, una topologia monodimensionale, perché ogni pacchetto può scegliere soltanto se andare a destra o a sinistra. La **griglia** o **mesh** (maglia, reticolato) della Figura 8.35(e) è un progetto bidimensionale, utilizzato in molti sistemi commerciali. È altamente regolare, facile da implementare al crescere delle dimensioni del sistema e il suo diametro cresce come la radice quadrata del numero di nodi. Una variante della griglia è il **toro** della Figura 8.35(f), cioè una griglia con le estremità connesse. Non solo ha una miglior tolleranza agli errori rispetto alla griglia, ma presenta anche un diametro inferiore, perché i vertici opposti possono ora comunicare in soli due passaggi.

Un'altra topologia diffusa è una struttura 3D con nodi nei punti le cui coordinate sono gli interi nell'intervallo che va da $(1, 1, 1)$ a (l, m, n). Ogni nodo ha sei vicini, due lungo ciascun asse. I nodi sui bordi hanno collegamenti che li collegano al bordo opposto, proprio come in un toro.

Il **cubo** della Figura 8.35(g) è una normale topologia tridimensionale. Nella figura illustriamo un cubo $2 \times 2 \times 2$, ma in generale si potrebbe avere un cubo $k \times k \times k$. La Figura 8.35(h) mostra un ipercubo a quattro dimensioni costruito a partire da due cubi mediante il collegamento dei nodi corrispondenti. Potremmo costruire un cubo a cinque dimensioni clonando la struttura della Figura 8.35(h) e collegando i nodi corrispondenti in modo da formare un blocco di quattro cubi. Per raggiungere le sei dimensioni possiamo replicare il blocco di quattro cubi e interconnettere i nodi corrispondenti, e così via. Un cubo n-dimensionale così formato si chiama un **ipercubo**. Molti computer paralleli usano questa topologia perché il suo diametro cresce linearmente con n. Detto altrimenti, il diametro è il logaritmo in base 2 del numero di nodi, perciò un ipercubo a dieci dimensioni di 1024 nodi ha un diametro di 10, il che garantisce proprietà di ritardo eccellenti. Si noti, come termine di paragone, il caso di 1024 nodi sistemati in una griglia 32×32: il loro diametro è 62, più di sei volte quello dell'ipercubo. Il prezzo pagato dall'ipercubo per avere un diametro inferiore è un maggiore fanout e quindi un maggiore numero di collegamenti (che accrescono il costo della rete). Malgrado ciò, l'ipercubo è una scelta comune a molti sistemi dalle prestazioni elevate.

Esistono multicomputer di tutte le forme e dimensioni, perciò è difficile darne una tassonomia chiara. Si possono identificare comunque due "stili" generali: gli MPP e i cluster. Studiamoli uno per volta.

8.4.2 MPP: processori massicciamente paralleli

La prima categoria che individuiamo è formata dagli **MPP** (*Massively Parallel Processor*, "processore massicciamente parallelo"): supercomputer enormi da svariati milioni di euro, usati per svolgere ingenti quantità di calcoli scientifici, ingegneristici o industriali, per la gestione di grandi numeri di transazioni al secondo o per l'immagazzinamento di dati (memorizzazione e gestione di database immensi). Inizialmente gli MPP venivano usati come supercomputer scientifici, ma adesso gran parte di loro è impiegata in ambienti commerciali. Queste macchine sono in un certo senso i successori dei poderosi mainframe degli anni '60 (ma la parentela è molto lontana, come se un paleontologo sostenesse che uno stormo di passeri discende dal *Tyrannosaurus Rex*). Si può dire che gli MPP hanno soppiantato al vertice della catena alimentare digitale le macchine SIMD, i supercomputer vettoriali e le unità di calcolo vettoriale.

Molte macchine MPP usano come processori CPU standard. È frequente l'uso di Pentium (Intel), di UltraSPARC (Sun) e di PowerPC (IBM). Ciò che distingue gli MPP è l'uso di una rete d'interconnessione brevettata, ad altissime prestazioni, progettata per trasferire i messaggi con poca latenza ed elevata larghezza di banda. Sono due proprietà importanti, perché nella stragrande maggioranza dei casi i messaggi sono piccoli (ben sotto i 256 byte), ma gran parte del traffico totale è dovuto ai messaggi grandi (più di 8 KB). Gli MPP vengono forniti con grandi quantità di software e librerie brevettati.

Un'altra caratteristica degli MPP è la loro enorme capacità di I/O. I problemi abbastanza grandi da meritare l'impiego di un MPP coinvolgono sempre enormi quantità di dati, spesso dell'ordine dei terabyte. I dati devono essere distribuiti tra molti dischi e devono essere trasferiti all'interno della macchina a grande velocità.

Infine, un'altra questione d'interesse circa gli MPP è la loro tolleranza agli errori. La presenza di migliaia di CPU rende inevitabili un certo numero di anomalie nel corso del tempo. Interrompere una computazione di 18 ore per l'insuccesso di una CPU è inaccettabile, specialmente se è prevedibile osservare una tale anomalia nel corso di una settimana. Perciò gli MPP più grandi sono sempre dotati di hardware e software speciali per il monitoraggio del sistema, per il rilevamento delle anomalie e il loro trattamento.

Sebbene sarebbe interessante studiare i principi generali alla base dei progetti MPP, in realtà va detto che non ci sono molti principi di base. In fondo gli MPP sono una raccolta di nodi di calcolo più o meno standard, collegati tramite un'interconnessione molto veloce scelta tra uno dei tipi già illustrati. In ragione di ciò, passiamo piuttosto allo studio di due esempi di MPP: BlueGene/L e Red Storm.

BlueGene

Come primo esempio di processore massicciamente parallelo prendiamo in esame il sistema BlueGene di IBM, un progetto concepito nel 1999 per la risoluzione di problemi che esigono molte risorse di calcolo, come quelli che sorgono nelle scienze biologiche. Per esempio, i biologi credono che la struttura tridimensionale di una proteina determini la sua funzionalità, eppure il calcolo della struttura spaziale di una piccola proteina a partire dalle leggi fisiche impiegava anni sui supercomputer del tempo. Ogni essere umano contiene più di mezzo milione di proteine, molte delle quali sono estremamente grandi, ed è noto che certi loro ripiegamenti errati (*misfolding*) sono responsabili di alcune malattie (come la fibrosi cistica). È chiaro che la determinazione della stereostruttura di tutte le proteine umane richiederebbe un incremento della capacità di calcolo mondiale di svariati ordini di grandezza, e la modellazione delle proteine è solo uno dei problemi per cui BlueGene è stato progettato. Esistono sfide ugualmente complesse nella dinamica molecolare, dei modelli climatici, in astronomia o anche nella modellazione finanziaria, che richiederebbero tutte un potenziamento dei supercomputer di diversi ordini di grandezza.

IBM crede così tanto nel mercato del supercalcolo massiccio che ha investito 100 milioni di dollari nel progetto e nella costruzione di BlueGene. Nel novembre 2001, il Livermore National Laboratory, gestito dal dipartimento statunitense per l'energia, ha firmato un contratto come primo cliente di un esemplare della famiglia BlueGene, chiamato **BlueGene/L**.

L'obiettivo del progetto non era solo quello di produrre il supercomputer MPP più veloce del mondo, ma anche di produrre l'esemplare più efficiente in termini di teraflop/dollaro, teraflop/watt e teraflop/m^3, cioè di costi, consumo energetico e dimensione. Per questa ragione IBM rifiutò la filosofia che ispirava gli MPP precedenti, cioè di usare i componenti più veloci che si potessero comprare. Al contrario, si decise di produrre un componente su misura con un intero sistema nel chip, che girasse a velocità modesta e che consumasse poca energia, in modo da produrre una macchina davvero grande e con un'alta densità di assemblaggio. Il primo chip venne alla luce nel giugno 2003. Il primo quarto del sistema BlueGene/L, costituito da 16.384 nodi, divenne pienamente operativo a partire dal novembre 2004 e, grazie ai suoi 71 teraflop/s, fu insignito del premio di supercomputer più veloce al mondo. Il consumo di soli 0,4 MWatt gli garantì anche il premio di supercomputer più efficiente della sua categoria, con 177,5 megaflop/watt. La consegna della parte restante del sistema, che lo porterebbe a una dimensione di 65.536 nodi di calcolo, era prevista per l'estate del 2005.

Il punto forte del sistema BlueGene/L è rappresentato dai suoi nodi, ciascuno costituito dal chip su misura illustrato nella Figura 8.36, formato da due core PowerPC 440 a 700 MHz. Il PowerPC 440 è un processore superscalare a pipeline e a doppia emissione, molto diffuso nei sistemi integrati. Ogni core ha un paio di unità in virgola mobile a doppia emissione, che possono emettere congiuntamente quattro istruzioni in virgola mobile per ogni ciclo. Le unità in virgola mobile sono state potenziate con l'aggiunta d'istruzioni di tipo SIMD che si rivelano spesso utili al calcolo scientifico vettoriale. Benché non sia proprio inefficiente, sicuramente non è un multiprocessore al vertice della gamma.

I due core di CPU sono identici per struttura, ma sono programmati diversamente: uno viene usato per il calcolo, l'altro per gestire la comunicazione tra i 65.536 nodi.

Nel chip sono presenti tre livelli di cache. Le cache L1 (primo livello) hanno 32 KB per le istruzioni e 32 per i dati. Non c'è alcuna coerenza tra le cache L1 delle due CPU perché i core PowerPC 440 non la supportavano e si è deciso di non modificarli. Le cache L2 (secondo livello) sono unificate e grandi 2 KB; non vengono usate come vere cache, ma come buffer per il prefetch. Ognuna fa snooping sull'altra cache e sono reciprocamente consistenti. La cache L3 (di terzo livello) ha 4 MB, è condivisa, unificata e consistente, e rifornisce le

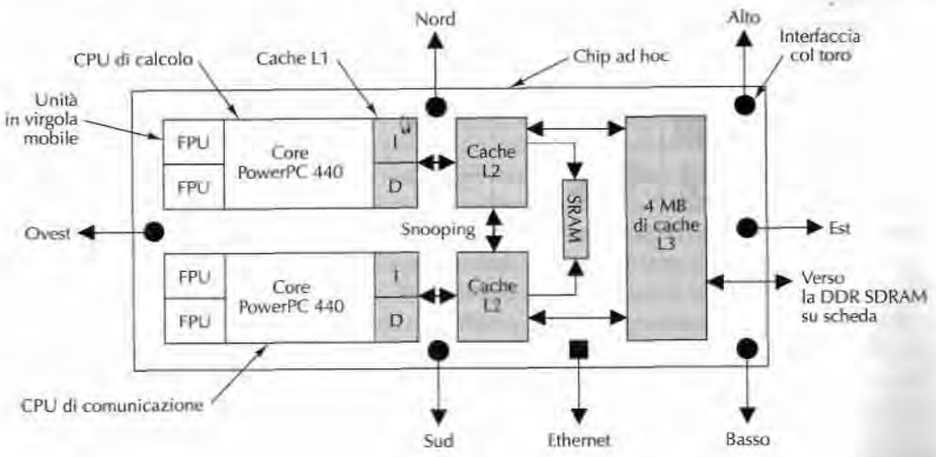


Figura 8.36 Il chip costruito per BlueGene/L.

cache L2. Un riferimento alla memoria che fallisce nella cache L1, ma che ha successo nella L2, impiega circa 11 cicli di clock. Un fallimento a livello 2 che ha successo a livello 3 impiega 28 cicli. Infine, un riferimento che non ha successo neanche nella cache L3 deve raggiungere la SDRAM principale e impiega 75 cicli.

Le cache L2 sono collegate a una piccola SRAM, a sua volta connessa all'interfaccia JTAG per il riavvio, il debugging e la comunicazione con l'host principale che gestisce lo stack di sistema e fornisce i semafori, le barriere e altre operazioni di sincronizzazione.

Al livello gerarchico successivo, IBM ha progettato su misura una scheda con due chip (Figura 8.36) e una RAM da 1 GB (le versioni future potranno contenerne 4 GB). Il chip e la scheda sono mostrati nelle Figure 8.37(a)-(b).

Le schede sono montate a innesto su una piastra che può ospitarne 16 per un totale di 32 chip (e quindi di 32 CPU di calcolo). Poiché ogni scheda contiene 1 GB di SDRAM, ogni piastra conteniene 16 GB di memoria. La Figura 8.37(c) illustra una piastra.

A livello successivo, le piastre vengono inserite in un armadio di base 60 × 90 cm. La metà superiore ospita 16 piastre e altre 16 trovano spazio nella metà inferiore, per un totale di 1024 CPU, una densità molto elevata. I due gruppi di 16 piastre sono separati da un commutatore che può estromettere un gruppo dal sistema per ragioni di manutenzione. La Figura 8.37(d) mostra un armadio.

Il sistema completo consiste in 64 armadi con 65.536 CPU di calcolo e altre 65.536 per la comunicazione, come rappresentato nella Figura 8.37(e). Grazie alle 131.072 CPU intere a doppia emissione e alle 262.144 CPU in virgola mobile a doppia emissione, il sistema potrebbe emettere fino a 786.432 istruzioni in un ciclo di clock. Tuttavia, una delle unità intere serve a rifornire le unità in virgola mobile e ciò riduce il numero d'istruzioni a 655.360 in ogni ciclo di clock, cioè a $4{,}6 \times 10^{14}$ istruzioni/s, il che lo rende comunque e di gran lunga il più potente sistema di calcolo mai costruito.

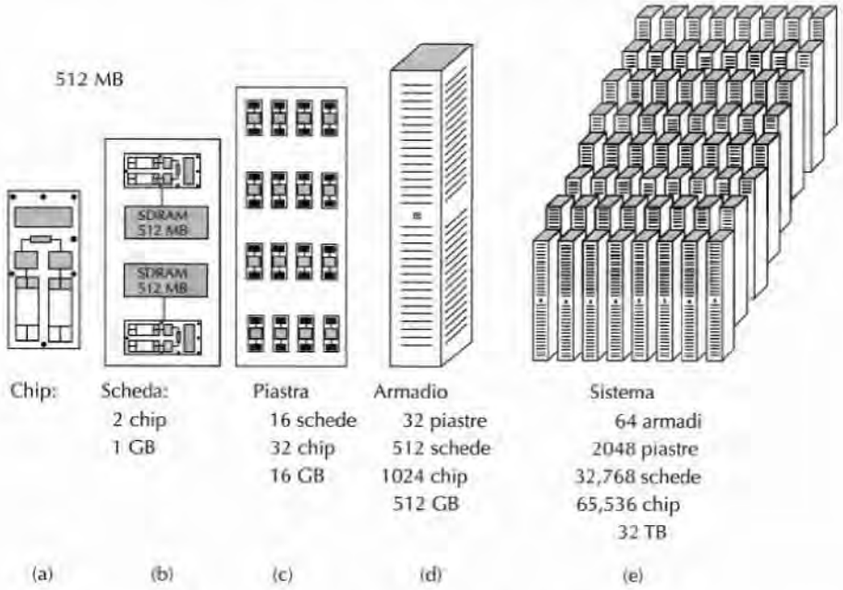


Figura 8.37 Componenti gerarchiche di BlueGene/L: (a) Chip (b) Scheda (c) Piastra (d) Armadio (e) Sistema.

Il sistema è un multicomputer nel senso che nessuna CPU ha accesso diretto alla memoria, fatta eccezione per i 512 MB residenti sulla sua scheda. Nessuna coppia di CPU condivide memoria comune. Inoltre, non si effettua paginazione a richiesta perché non ci sono dischi locali da cui paginare. Il sistema dispone invece di 1024 nodi di I/O collegati a dischi e ad altri dispositivi periferici.

Tutto considerato, a dispetto delle sue dimensioni esagerate, il sistema è abbastanza semplice e introduce poca tecnologia, a eccezione della capacità di assemblaggio ad alta densità. La scelta della semplicità non è stata accidentale, ma è frutto dell'aver perseguito come obiettivi primari l'elevata affidabilità e reperibilità dei componenti. Di conseguenza, è stata dedicata molta cura ingegneristica all'alimentazione elettrica, alle ventole, al raffreddamento e alla cablatura, allo scopo di raggiungere un tempo medio di 10 giorni tra l'occorrenza di due anomalie successive.

La connessione di tutti i chip richiede una rete d'interconnessione scalabile e dalle elevate prestazioni. Il progetto impiegato è un toro 64 × 32 × 32. Perciò ogni CPU ha bisogno di sei collegamenti: due verso le CPU che si trovano logicamente in alto o in basso rispetto a essa, due verso est e ovest, due verso nord e sud. I sei collegamenti sono etichettati quindi come alto, basso, est, ovest, nord e sud, come evidenziato nella Figura 8.36. Ogni armadio è un toro 8 × 8 × 16. Quattro paia di armadi della stessa riga formano un toro 8 × 32 × 32. Infine, le otto righe di armadi formano il toro 64 × 32 × 32.

Dunque tutti i collegamenti sono diretti da nodo a nodo e operano a 1,4 Gbps. Visto che ciascuno dei 65.536 nodi ha tre collegamenti verso nodi di numero "maggiore", uno lungo ciascuna dimensione, la larghezza di banda totale del sistema è di 275 terabit/s. Il contenuto informativo di questo libro è pari a circa 300 milioni di bit, inclusa la grafica in formato incapsulato PostScript, perciò BlueGene sarebbe in grado di trasferire 900.000 copie di questo libro in un secondo. Dove sarebbero distribuite e chi le potrebbe richiedere è una domanda che lasciamo al lettore come esercizio.

La comunicazione nel toro si svolge sotto forma d'instradamento **cut through virtuale**. Si tratta di una tecnica abbastanza simile alla commutazione di pacchetto store-and-forward, con la differenza che i pacchetti non vengono memorizzati prima di essere inoltrati. Non appena un byte raggiunge un nodo, può essere inoltrato immediatamente al nodo successivo lungo il cammino, addirittura prima che giunga l'intero pacchetto cui appartiene. Sono possibili sia l'instradamento dinamico (adattivo), sia deterministico (fisso). Una piccola parte dell'hardware del chip è destinato all'implementazione del cut through virtuale.

Oltre al toro principale usato per il trasferimento dati, sono presenti altre quattro reti di comunicazione. La seconda è una rete ad albero, *combining network*, che riunisce tutti i nodi. Infatti molte delle operazioni eseguite nei sistemi altamente paralleli come BlueGene/L hanno bisogno della partecipazione di tutti i nodi. Si consideri, per esempio, la ricerca del minimo in un insieme di 65.536 valori, ciascuno contenuto in un nodo. La rete in questione riunisce tutti i nodi in un albero e può essere usata per il calcolo del minimo: quando un nodo riceve i valori provenienti dai nodi a lui inferiori, sceglie il più piccolo tra questi e il proprio valore e lo inoltra lungo l'albero verso l'alto. Così facendo, il nodo radice dell'albero

viene raggiunto da un numero di pacchetti molto inferiore a quanti ne riceverebbe se tutti i 65.536 nodi gli spedissero un messaggio direttamente.

La terza rete è usata per le barriere globali e gli interrupt. Alcuni algoritmi lavorano in più passi e richiedono che ogni nodo attenda finché tutti gli altri hanno completato la fase in esecuzione prima di entrare nella successiva. La rete di barriere consente la definizione di queste fasi via software e fornisce uno strumento per la sospensione di tutte le CPU che hanno terminato una fase, fino al completamento della fase da parte di tutte le altre CPU; solo a questo punto vengono liberate dall'attesa. Anche gli interrupt viaggiano lungo questa rete.

La quarta e la quinta rete usano entrambe una Gigabit Ethernet. La prima collega i nodi di I/O ai server di file, che sono esterni a BlueGene/L, e attraverso questi a Internet. L'altra serve al debugging del sistema.

Ogni nodo di calcolo e comunicazione esegue un piccolo kernel progettato su misura, che supporta un solo utente e un solo processo. Questo processo contiene due thread, uno per ogni CPU del nodo. La semplicità della struttura è intesa a garantire prestazioni e affidabilità elevate.

Per incrementare l'affidabilità, il software applicativo può richiamare una procedura di libreria per inserire un punto di controllo (*checkpoint*). Una volta spediti in rete tutti i messaggi in uscita, si può stabilire e memorizzare un punto di controllo globale di modo che, a seguito di un'anomalia del sistema, sia possibile far ripartire l'esecuzione dal punto di controllo invece che dall'inizio. I nodi di I/O utilizzano il tradizionale sistema operativo Linux e supportano processi multipli. Si possono trovare maggiori informazioni su BlueGene/L in (Adiga et al., 2002; Almasi et al., 2003a, 2003b; Blumrich et al., 2005).

Red Storm

Come secondo esempio di MPP, prendiamo in considerazione la macchina Red Storm (detta anche il "Martello di Thor") che si trova presso Sandia National Laboratory. Questo istituto è gestito dalla Lockheed Martin e svolge attività per conto del dipartimento statunitense dell'energia, sia di natura riservata (segreta) sia pubblica. Parte del lavoro riservato concerne il progetto e la simulazione di armi nucleari, che richiedono molta potenza di calcolo.

Sandia è un soggetto attivo da molti anni e nel passato ha avuto a disposizione numerosi supercomputer di prima caratura. Per alcuni decenni ha prediletto i supercomputer vettoriali, ma alla lunga gli MPP sono risultati più convenienti sia dal punto di vista tecnologico, sia economico. Nel 2002 ASCI Red, l'allora supercomputer MPP di Sandia, cominciava a scricchiolare. Era costituito da 9460 nodi, ma in totale disponeva di soltanto 1,2 TB di RAM e 12,5 TB di spazio su disco, e il sistema poteva macinare appena 3 teraflop/s. Perciò nell'estate del 2002 Sandia ha scelto Cray Research, un costruttore di supercomputer di vecchia data, per far costruire il sostituto dell'ASCI Red.

Il rimpiazzo venne consegnato nell'agosto 2004, dopo un lasso di tempo eccezionalmente contenuto per il progetto e l'implementazione di una macchina così grande. La ragione della celerità della sua progettazione e consegna è che Red Storm è composto quasi interamente da parti già pronte all'uso, fatta eccezione per un chip per l'instradamento progettato su misura.

La CPU scelta per Red Storm è Opteron di AMD. Opteron presenta numerose caratteristiche chiave che ne fanno la scelta migliore. La prima è che può funzionare in tre modalità operative. In modalità *legacy* Opteron esegue programmi standard per il Pentium senza bisogno di modifiche; in modalità di *compatibilità*, il sistema operativo funziona a 64 bit e può indirizzare 2^{64} parole di memoria (ma i programmi applicativi girano a 32 bit); in modalità a 64 bit, infine, l'intera macchina funziona a 64 bit e tutti i programmi possono utilizzare lo spazio degli indirizzi di 64 bit. In quest'ultimo caso si possono affiancare nell'esecuzione programmi a 32 e a 64 bit, il che predispone il sistema ad aggiornamenti futuri.

La seconda caratteristica chiave di Opteron è la sua attenzione al problema della larghezza di banda della memoria. Negli ultimi anni la velocità delle CPU è cresciuta molto più rapidamente rispetto alla larghezza di banda della memoria, rendendo i fallimenti di cache di secondo livello un grosso fattore di penalizzazione delle prestazioni. AMD ha integrato il controllore della memoria all'interno di Opteron di modo che possa girare alla stessa velocità del clock del processore e non più alla velocità del bus della memoria, migliorando le prestazioni di memoria. Il controllore può gestire otto DIMM di 4 GB ciascuna, per un totale di 32 GB di memoria per ogni Opteron. Nei sistemi Red Storm ogni Opteron ha solo 2-4 GB di memoria, però si può star sicuri che in futuro ne verrà aggiunta altra, visto che la memoria diventa sempre più economica. Un altro possibile miglioramento è l'adozione dei chip Opteron con due core, che raddoppierebbe la potenza grezza di calcolo.

Ogni Opteron ha un proprio processore di rete dedicato, **Seastar**, progettato su misura e prodotto da IBM. Si tratta di un componente critico perché quasi tutto il traffico di dati tra processori passa lungo la rete Seastar. Se non ci fossero questi chip progettati appositamente per garantire un'interconnessione ad altissima velocità, il sistema si troverebbe spesso impantanato nella grande quantità di dati.

I processori Opteron sono prodotti commerciali facilmente reperibili e pronti all'uso, ma le parti di assemblaggio di Red Storm sono costruite su misura. Ogni piastra di Red Storm (Figura 8.38) contiene quattro Opteron, 4 GB di RAM, quattro Seastar, un pro-

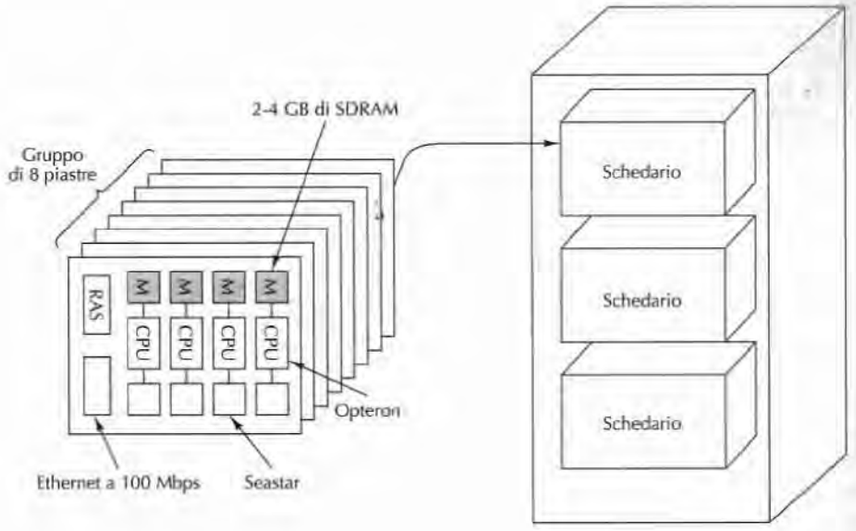


Figura 8.38 Assemblaggio dei componenti di un Red Storm.

cessore RAS (*Reliability-Availability-Service*, "affidabilità-reperibilità-servizio") e una scheda Ethernet a 100 Mbps.

Le piastre sono raggruppate a otto a otto e collegate a una scheda madre inserita in uno schedario. Ogni armadio contiene tre sportelli di schede per un totale di 96 Opteron, oltre ai necessari dispositivi di alimentazione e ventilazione. L'intero sistema è formato da 108 armadi per i nodi di calcolo, per un totale di 10.368 Opteron con 10 TB di SDRAM. Ogni CPU ha accesso esclusivamente alla propria SDRAM: non c'è alcuna memoria condivisa. La potenza di calcolo teorica del sistema è 41 teraflop/s.

L'interconnessione tra le CPU Opteron è realizzata tramite i router Seastar (uno per ogni CPU), collegati per mezzo di un toro di dimensioni 27 × 16 × 24 (a ogni intersezione della rete c'è un chip Seastar). Ogni Seastar ha sette collegamenti bidirezionali da 24 Gbps che sono diretti verso nord, est, sud, ovest, alto, basso e verso il processore Opteron. Il tempo di transito tra due punti adiacenti nella rete è 2 μs, e bastano 5 μs per attraversare tutto l'insieme dei nodi di calcolo. C'è anche una seconda rete Ethernet a 100 Mbps per servizio e manutenzione.

Oltre ai 108 armadi di calcolo, il sistema comprende 16 armadi per i processori di I/O e di servizio. Ciascuno di loro contiene 32 Opteron le cui CPU sono così suddivise: 256 per l'I/O, 256 di servizio. Lo spazio rimanente è occupato dai dischi, impostati come RAID 3 e RAID 5, e a ciascuno di loro è associato un disco di parità e uno di ricambio. Lo spazio totale su disco è di 240 TB, mentre la larghezza di banda complessiva dei dischi è di 50 GB/s.

Il sistema è suddiviso in due sezioni, una riservata e una no, collegate tramite commutatori e che possono quindi essere agganciate o sganciate meccanicamente. Entrambe le parti contengono sempre almeno 2688 CPU di calcolo. Le rimanenti 4992 CPU di calcolo possono essere commutate su una qualsiasi delle due sezioni, come illustrato nella Figura 8.39. Le 2688 CPU riservate sono Opteron con 4 GB di RAM, mentre le rimanenti hanno 2 GB. I processori di I/O e di servizio sono ripartiti nelle due sezioni.

Il tutto è ospitato all'interno di un edificio di 2000 m^2 costruito per l'occasione. L'edificio è stato concepito in modo da poter ospitare un sistema Red Storm che possa essere potenziato in futuro fino a 30.000 CPU. I nodi di calcolo assorbono 1,6 MW di potenza, e un altro MW è assorbito dai dischi. Se si conteggiano anche i consumi dei sistemi di aerazione e raffreddamento, le apparecchiature assorbono complessivamente 3,5 MW.

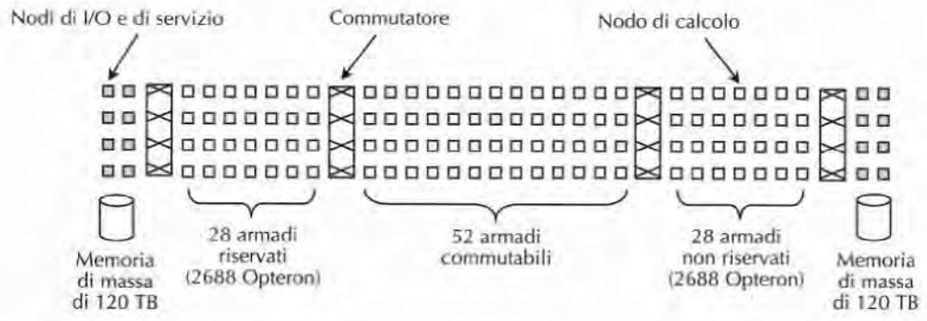


Figura 8.39 Visione dall'alto di Red Storm.

Il costo di hardware e software è stato di 90 milioni di dollari, l'edificio e il raffreddamento sono costati altri 9 milioni di dollari. Ovviamente parte dei costi è da considerarsi una tantum, in quanto dovuti alla progettazione e alla ingegnerizzazione. Se voleste ordinare una copia di Red Storm dovreste pensare a una cifra dell'ordine dei 60 milioni di dollari. Cray ha intenzione di produrre una versione più contenuta del sistema, che si chiamerà X3T.

I nodi di calcolo eseguono un kernel leggero che si chiama **catamount**. I nodi di I/O e quelli di servizio eseguono la versione vanilla di Linux con l'aggiunta del supporto di MPI (ritorneremo sull'argomento nel corso del capitolo). I nodi RAS eseguono una versione di Linux ridotta al minimo. Red Storm può eseguire quasi tutto il software scritto per ASCI Red, compresi i programmi di allocazione per le CPU, gli schedulatori, le librerie MPI, le librerie matematiche e anche i programmi applicativi.

Per un sistema così grande l'affidabilità è essenziale. Ogni piastra ha un processore RAS per la manutenzione, più altre funzionalità hardware speciali. L'obiettivo è quello di raggiungere un MTBF (*Mean Time Between Failures*), ovvero un tempo medio tra due guasti di 50 ore. ASCI Red poteva vantare un MTBF di 900 ore, ma subiva il crollo del sistema operativo circa ogni 40 ore. Benché il nuovo hardware sia molto più affidabile del vecchio, il software resta il punto debole del sistema.

Per ulteriori informazioni su Red Storm si faccia riferimento a (Brightwell et al., 2005).

Confronto tra BlueGene/L e Red Storm

Red Storm e BlueGene/L sono per certi versi paragonabili, ma sotto altri aspetti restano molto differenti, perciò è interessante elencare i loro parametri fondamentali (Figura 8.40).

Le due macchine sono state costruite quasi contemporaneamente, perciò le loro differenze non sono attribuibili a livello tecnologico, ma alle diverse visioni dei progettisti e, in una certa misura, anche alle differenze tra i due costruttori, IBM e Cray. BlueGene/L è stato progettato sin dall'inizio come una macchina commerciale che IBM spera di vendere in gran numero alle aziende biotecnologiche, farmaceutiche, e così via. Red Storm è nato da un contratto speciale con Sandia, anche se Cray pensa di venderne comunque una versione ridotta.

La visione di IBM è chiara: combinare core già esistenti in chip ideati su misura che possono essere prodotti con bassi costi unitari. I chip girano a bassa velocità e sono facilmente componibili in gran numero mediante una rete di comunicazione di velocità modesta. La visione di Sandia è altrettanto chiara, ma diversa: usare una potente CPU a 64 bit già esistente, progettare su misura un chip per l'instradamento estremamente veloce e collegare al tutto tanta memoria, ottenendo così un nodo di gran lunga più potente di quello di BlueGene/L; dunque basterà un numero di nodi inferiore e la comunicazione tra di loro potrà essere più veloce.

Queste scelte influenzano direttamente l'assemblaggio dei componenti. IBM ha raggiunto una maggiore densità grazie all'inserimento di un processore e di un router all'interno di un solo chip: 1024 CPU/armadio. Viceversa, Sandia ha scelto di usare senza modifiche un chip di CPU già esistente e di equipaggiare ogni nodo con 2-4 GB di RAM, perciò è riuscita ad alloggiare solo 96 CPU in ogni armadio. In ragione di ciò, Red Storm occupa più superficie e assorbe più potenza di BlueGene/L.

Nell'esotico mondo dei laboratori di ricerca, il traguardo si misura in termini di prestazioni. Da questo punto di vista BlueGene/L vince per 71 a 41 Tflop/s, ma Red Storm è

Elemento	BlueGene/L	Red Storm
CPU	PowerPC a 32 bit	Opteron a 64 bit
Clock	700 MHz	2GHz
CPU di calcolo	65.536	10.368
CPU per piastra	32	4
CPU per armadio	1024	96
Armadi di calcolo	64	108
Teraflop/s	71	41
Memoria per CPU	512 MB	2-4 GB
Memoria complessiva	32 TB	10 TB
Router	PowerPC	Seastar
Numero di router	65.536	10.368
Interconnessione	Toro 3D 64 × 32 × 32	Toro 3D 27 × 16 × 24
Altre reti	Gigabit Ethernet	Fast Ethernet
Ripartibile	No	Sì
SO di calcolo	Su misura	Su misura
SO di I/O	Linux	Linux
Costruttore	IBM	Cray Research
Costo elevato	Sì	Sì

Figura 8.40 Confronto tra BlueGene/L e Red Storm.

espandibile e grazie all'aggiunta di altri 10.368 Opteron (per esempio passando ai chip con due core) potrebbe raggiungere gli 82 Tflop/s. IBM potrebbe rispondere spingendo un po' sul pedale del clock (una frequenza di 700 MHz non è proprio il massimo della tecnologia più avanzata). In breve, i supercomputer MPP non hanno ancora avvicinato i limiti fisici imposti dalla loro architettura e continueranno a crescere negli anni a venire.

8.4.3 Cluster

La seconda categoria di multicomputer è costituita dai **cluster di computer** (Anderson et al., 1995; Martin et al., 1997). Si tratta in genere di centinaia o migliaia di PC o di workstation (stazioni di lavoro) collegate per mezzo di schede di rete reperibili sul mercato. La differenza tra un MPP e un cluster è analoga a quella tra un mainframe e un PC: entrambi hanno una CPU, un po' di RAM, dischi, un sistema operativo, ecc., solo che nei mainframe tutto è più veloce (a parte forse il sistema operativo). Ciononostante sembrano diversi e sono infatti usati e gestiti in modi differenti. Lo stesso accade agli MPP e ai cluster.

Storicamente, la caratteristica che ha sempre distinto gli MPP è l'interconnessione ad alta velocità, ma l'arrivo sul mercato di collegamenti veloci e pronti all'uso ha cominciato a colmare il divario. È probabile che i cluster spingano gli MPP verso nicchie sempre più ristrette, proprio come i PC hanno fatto con i mainframe, trasformandoli in oggetti esoterici

per specialisti. La nicchia principale per gli MPP è quella dei supercomputer molto costosi cui si richiedono le massime prestazioni, ma il cui prezzo è praticamente irraggiungibile.

Anche se esistono molti tipi di cluster, le tipologie dominanti sono due: i cluster centralizzati e quelli decentralizzati. I primi sono cluster di workstation o di PC montati su grossi scaffali in un unico ambiente. Alle volte sono assemblati in modo ancora più compatto per ridurre le dimensioni e la lunghezza dei cavi. In genere le macchine sono omogenee e non hanno alcuna periferica, a parte la scheda di rete ed eventualmente alcuni dischi. Gordon Bell, il progettista del PDP-11 e di VAX, ha definito queste macchine **headless workstation** ("stazioni di lavoro senza capo", nel senso che non hanno un proprietario). Saremmo stati tentati di definirle COW senza capo, ma ci siamo trattenuti per il timore di sacrificare troppi bovini[5].

I cluster decentralizzati sono formati da workstation e PC diffusi all'interno di un edificio o di un intero campus. Molte macchine restano inoperose per diverse ore al giorno, specie di notte, e sono spesso collegate tramite una LAN. Si tratta in genere di macchine eterogenee e dotate di un ricco equipaggiamento di periferiche, anche se un cluster con 1024 mouse non è per nulla migliore di uno che ne sia privo. Quel che più conta è che molti proprietari di computer sono assai gelosi delle loro macchine e guardano con sospetto gli astronomi che vogliono usarle per simulare il big bang. L'uso di workstation inoperose per formare un cluster richiede necessariamente la capacità di far migrare un compito da una macchina a un'altra quando il proprietario ne reclama l'uso. La migrazione dei compiti è possibile, ma complica il software.

I cluster sono spesso oggetti di piccole dimensioni, che vanno da una dozzina a 500 PC. È però possibile costruirne di molto grandi a partire da comuni PC. È proprio ciò che ha fatto Google e in un modo molto interessante, che ora esaminiamo.

Google

Google è un motore di ricerca molto in voga per la selezione d'informazioni in Internet. La sua popolarità è dovuta almeno in parte all'interfaccia semplice e al tempo di risposta rapido, ma il suo funzionamento è tutt'altro che semplice. Dal punto di vista di Google, il problema è trovare, indicizzare e memorizzare l'intero World Wide Web (più di 8 miliardi di pagine e 1 miliardo di immagini), riuscire a effettuare ricerche al suo interno in meno di mezzo secondo e gestire migliaia di interrogazioni al secondo che provengono da tutto il mondo, ventiquattrore al giorno. Per di più deve restare sempre attivo, anche in caso di terremoti, black out elettrici, interruzioni delle linee di comunicazione, anomalie hardware e bachi software. Tutto ciò deve essere realizzato naturalmente al minor costo possibile. La costruzione di un clone di Google sicuramente non è un esercizio per il lettore.

Come funziona Google? Per cominciare, Google gestisce numerosi centri dati in diversi paesi del mondo. Questa soluzione serve non solo a fornire un backup dei dati nel caso un centro venisse inghiottito da un terremoto, ma anche a smistare i collegamenti: quando un utente digita www.google.com il suo IP viene esaminato e la stringa viene tradotta nell'indirizzo del centro più vicino, e a quell'indirizzo il software di navigazione spedisce la richiesta.

---

[5] L'acronimo COW (*Cluster Of Workstations*) è anche la parola inglese che designa l'animale "mucca" (*N.d.T.*).

Ogni centro dati dispone di almeno una connessione in fibra ottica OC-48 (2,488 Gbps) con Internet, tramite cui riceve le interrogazioni e spedisce le risposte, oltre a una connessione di backup OC-12 (622 Mbps) verso un secondo fornitore di telecomunicazioni, nel caso il primo smettesse di funzionare. Ogni centro dati è equipaggiato con un gruppo di continuità e con generatori diesel di emergenza, per garantire la continuazione del servizio in caso di black out. Perciò Google è in grado di funzionare anche in concomitanza di una calamità naturale di grosse proporzioni, sebbene a prestazioni ridotte.

Per capire la scelta architetturale di Google, è utile descrivere brevemente l'elaborazione di un'interrogazione (*query*) che perviene al centro dati di pertinenza. Dopo l'arrivo (passo 1 nella Figura 8.41), il bilanciatore di carico instrada l'interrogazione verso uno dei tanti gestori d'interrogazioni (2) e da lì prosegue in parallelo al correttore ortografico (3) e al server di pubblicità (4). Quindi le diverse parole di ricerca sono cercate in parallelo nei server degli indici (5) che contengono un elemento per ogni parola presente nel Web. Ogni elemento contiene una lista di tutti i documenti (pagine web, file PDF, presentazioni PowerPoint, e così via) che contengono la parola, ordinati secondo il loro *page rank* (la posizione che ogni pagina detiene nella classifica globale del Web). Il page rank è calcolato con una formula complicata (e segreta), in cui giocano un ruolo predominante il numero di collegamenti entranti nella pagina e il rank delle pagine da cui provengono.

Al fine di incrementare le prestazioni, gli indici sono divisi in **shard** ("frammenti, schegge") sui quali è possibile effettuare la ricerca in parallelo. Almeno dal punto di vista concettuale, lo shard 1 contiene tutte le parole indicizzate, ciascuna seguita dagli ID dei primi *n* documenti (nella classifica delle pagine) contenenti quella parola. Lo shard 2 contiene tutte le parole e gli ID degli *n* documenti successivi (nella classifica delle pagine) e così via. Quando il Web sarà cresciuto ulteriormente, si potrà dividere ogni shard in un insieme di shard di *k* parole ciascuno, per ottenere un maggiore parallelismo di ricerca.

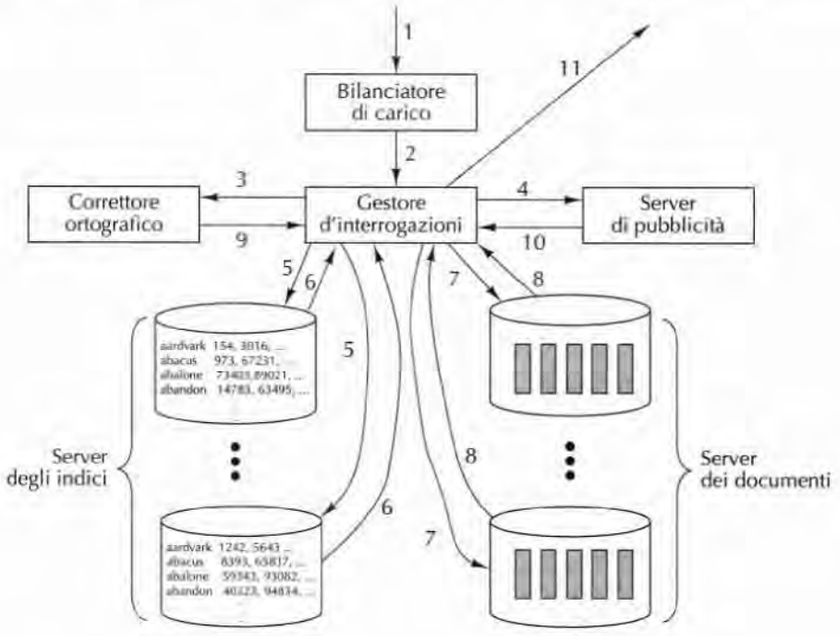


Figura 8.41 Gestione di un'interrogazione Google.

I server degli indici restituiscono un insieme di identificativi di documento (6) che sono poi combinati secondo le relazioni booleane specificate dall'interrogazione. Per esempio, se si è ricercato + digitale + bradipo + danza, solo i documenti che appaiono in tutti e tre gli insiemi sono usati al passo successivo (7), in cui si accede ai documenti stessi per ricavarne i titoli, gli URL e gli estratti di testo che circondano i termini ricercati. I server dei documenti contengono molte copie dell'intero Web in ciascun centro dati (al momento si parla di centinaia di terabyte). I documenti sono suddivisi a loro volta in shard per aumentare il parallelismo di ricerca. Anche se un'interrogazione non richiede la lettura dell'intero Web (e neanche delle decine di terabyte dei server degli indici), normalmente comporta comunque un'elaborazione dell'ordine dei 100 MB.

I risultati vengono quindi rispediti al gestore delle interrogazioni (8) che riordina le pagine secondo il loro page rank. Se sono stati rilevati potenziali errori di digitazione (9) questi vengono resi noti e infine vengono aggiunti annunci pubblicitari attinenti alla ricerca (10). La vendita di specifici termini di ricerca (per esempio "hotel" o "videocamera") cui associare annunci pubblicitari è la sorgente dei proventi di Google. Infine, i risultati sono formattati in HTML (*HyperText Markup Language*) e spediti all'utente richiedente sotto forma di pagina web.

A questo punto possiamo esaminare l'architettura di Google. La maggior parte delle aziende, di fronte alla necessità di gestire un database enorme, comprerebbe l'attrezzatura più grande, veloce e affidabile che si trova sul mercato. Google ha fatto l'esatto opposto: ha comprato un gran numero di PC economici dalle prestazioni modeste e li ha usati per costruire il più grande cluster del mondo, fatto solo di componenti già disponibili sul mercato. Il principio guida alla base di questa decisione è semplice: ottimizzare il rapporto costi/prestazioni.

La logica della decisione è di natura economica: i PC sono a buon mercato, a differenza dei server di fascia alta e, soprattutto, dei multiprocessori di grandi dimensioni. Un server di fascia alta potrebbe offrire prestazioni due o tre volte superiori a quelle di un desktop PC di fascia media, a un prezzo che è in genere dalle cinque alle dieci volte maggiore, perciò la sua scelta non è conveniente.

Ovviamente i PC economici sono soggetti a più anomalie dei server che si trovano ai vertici della gamma, ma anche questi ultimi incappano alle volte in qualche malfunzionamento. Sin dall'inizio Google è stato progettato per funzionare su hardware fallibile, indipendentemente dal tipo di attrezzatura utilizzata. Una volta scritto software tollerante agli errori, non importa molto se il tasso di errore è dello 0,5% o del 2% all'anno, le anomalie vanno trattate in ogni caso. Nell'esperienza di Google, il 2% dei PC mostra almeno un'anomalia nel corso di un anno; più della metà dei malfunzionamenti sono dovuti a dischi difettosi, mentre i restanti sono attribuibili prevalentemente agli alimentatori elettrici e ai chip di RAM. Le CPU già collaudate non incappano mai in errori. In realtà la maggior parte dei crolli di sistema non è dovuta per nulla all'hardware, ma al software. La prima reazione a un crollo è riavviare il sistema, il che spesso risolve il problema (l'equivalente elettronico del consiglio del dottore: "prenda un'aspirina e vada a letto").

Un tipico PC moderno di Google dispone di un Pentium a 2 GHz, 512 MB di RAM e un disco di 80 GB circa, il tipo di calcolatore che comprerebbe una nonna per controllare la posta elettronica, di tanto in tanto. L'unica componente aggiuntiva è un chip Ethernet molto economico e per niente sofisticato. I PC sono impilati a gruppi di 40 in scaffali alti poco più di mezzo metro. Ogni scaffale ospita due gruppi, uno sul davanti, uno sul retro, per un totale di 80 PC. I PC dello stesso scaffale sono collegati tramite una Ethernet, il cui commutatore è contenuto nello scaffale. Anche gli scaffali di un centro dati sono collegati tramite Ethernet, con due commutatori ridondanti per far fronte alle anomalie di commutazione.

La Figura 8.42 riporta lo schema di un tipico centro dati di Google. La fibra OC-48 ad alta larghezza di banda, per le interrogazioni in entrata, viene instradata verso i commutatori Ethernet a 128 porte, come anche la fibra OC-12 di backup. Le fibre in entrata usano schede speciali e non impegnano alcuna delle 128 porte Ethernet. Ogni scaffale ha quattro collegamenti Ethernet, due verso il commutatore di sinistra, due verso quello di destra. Mediante questo schema il sistema può sopravvivere alla rottura di uno dei due commutatori. Poiché ogni scaffale ha quattro connessioni con i commutatori (due provenienti dai 40 PC sul davanti, due dai 40 sul retro), ci vogliono quattro interruzioni di collegamenti,

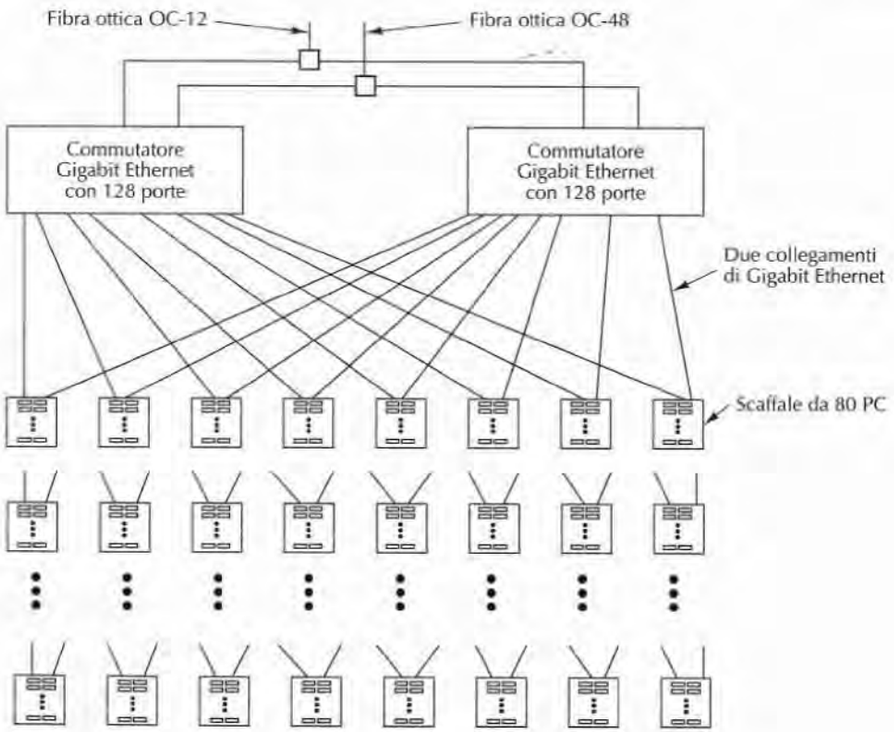


Figura 8.42 Un tipico cluster di Google.

oppure due interruzioni e una rottura di un commutatore, per disconnettere uno scaffale. I due commutatori a 128 porte possono connettere al massimo 64 scaffali con quattro collegamenti ciascuno. Dato che ogni scaffale ospita 80 PC, un centro dati può contenere fino a 5120 PC. Tuttavia gli scaffali possono contenere un numero di PC diverso da 80 e i commutatori possono avere più o meno di 128 porte. I numeri che abbiamo fornito sono solo quelli di un tipico cluster di Google.

Anche la densità di potenza è un fattore critico. Un PC comune assorbe 120 watt, uno scaffale 10 kilowatt. Uno scaffale ha bisogno di 3 m^2 perché ci sia spazio sufficiente alla manutenzione (installazione e rimozione di PC) e al sistema di condizionamento d'aria. Questi parametri inducono una densità di potenza maggiore di 3000 watt/m^2, mentre gran parte dei centri dati sono progettati per 600-1200 watt/m^2, dunque si rendono necessarie speciali misure per il raffreddamento degli scaffali.

Google ha acquisito tre conoscenze a proposito della gestione di server web di dimensioni enormi, che vale la pena ripetere.

1. I componenti sono soggetti ad anomalie, perciò bisogna prenderle in considerazione a priori.
2. Duplicare tutti i componenti aumenta la produttività e la disponibilità di risorse.
3. Conviene ottimizzare il rapporto prezzo/prestazioni.

Il primo punto afferma che bisogna progettare software tollerante agli errori. Anche se si dispone dell'attrezzatura migliore in assoluto, la presenza di un numero enorme di componenti implica necessariamente qualche malfunzionamento e il software deve essere pronto a gestire la situazione. Non importa se si verificano 1 o 2 anomalie alla settimana, il software deve essere in grado di fronteggiare i malfunzionamenti di sistemi di queste dimensioni.

Il secondo punto specifica che l'hardware e il software devono essere molto ridondanti. Così facendo non solo si migliorano le proprietà di tolleranza agli errori, ma anche la produttività (il *throughput*, cioè il volume di dati elaborati per unità di tempo). Nel caso di Google, PC, dischi, cavi e commutatori sono tutti duplicati più volte. Inoltre, anche all'interno dello stesso centro dati esistono molteplici copie di indici e documenti.

Il terzo punto è una conseguenza dei primi due. Se un sistema è stato progettato correttamente per trattare le anomalie, è un errore comprare componenti costosi come le unità RAID con dischi SCSI. Anche queste esibirebbero anomalie, e spendere 10 volte di più per dimezzare il tasso di errore è una cattiva idea. È meglio comprare hardware in quantità 10 volte maggiore e trattare gli errori quando si presentano. Inoltre, la presenza di hardware duplicato garantisce prestazioni migliori quando tutto funziona correttamente.

Facciamo riferimento a (Barroso et al., 2003; Ghemawat et al., 2003) per maggiori informazioni su Google.

8.4.4 Software di comunicazione per multicomputer

La programmazione di un multicomputer richiede software speciale, spesso di libreria, per la gestione della comunicazione e della sincronizzazione tra processi. In questo paragrafo spendiamo qualche parola in proposito. Gli MPP e i cluster possono eseguire per buona parte gli stessi pacchetti software, perciò le applicazioni possono essere portate facilmente da una piattaforma all'altra.

I sistemi a scambio di messaggi eseguono due o più processi in modo indipendente l'uno dall'altro. Per esempio, un processo potrebbe produrre alcuni dati e uno o più altri processi potrebbero consumarli. Non c'è alcuna garanzia che il destinatario (o i destinatari) sia pronto a trattare i dati quando questi sono disponibili presso il mittente, dal momento che ciascuno di loro esegue il proprio programma.

Molti dei sistemi a scambio di messaggi forniscono le primitive `send` e `receive` (spesso sotto forma di chiamate di libreria), ma ci sono possibili ulteriori possibilità. Le tre varianti principali sono:

1. scambio sincrono di messaggi
2. scambio di messaggi bufferizzato
3. scambio di messaggi non bloccante.

Lo **scambio sincrono di messaggi** prevede che, se il mittente esegue una `send` e il destinatario non ha ancora eseguito una `receive`, il mittente si blocca e resta sospeso finché il destinatario non esegue la `receive`, e solo a questo punto il messaggio viene copiato. Alla restituzione del controllo al mittente dopo la chiamata, questi è sicuro che il messaggio è stato spedito e ricevuto correttamente. Questo metodo ha una semantica semplice e non richiede l'uso di buffer. Presenta però la forte limitazione di sospendere il mittente fino alla consegna del messaggio al destinatario e della rispettiva ricevuta al mittente.

Nello **scambio di messaggi bufferizzato**, se un messaggio viene spedito prima che il destinatario sia pronto, il messaggio viene posto in un buffer da qualche parte, per esempio in una mailslot, finché il destinatario non lo riceve. In tal modo il mittente può proseguire la sua esecuzione dopo una `send`, anche se il destinatario è occupato in un'altra attività. Dal momento che il messaggio è stato effettivamente spedito, il mittente è libero di riutilizzare il buffer di messaggi immediatamente. Questo schema riduce il tempo di attesa per il mittente; infatti, il mittente è libero di proseguire l'esecuzione non appena il sistema abbia spedito il messaggio. Tuttavia, il mittente non ha alcuna garanzia che il messaggio sia stato ricevuto correttamente. Anche in presenza di comunicazione affidabile, potrebbe sempre accadere che l'esecuzione del destinatario si sia interrotta prima di aver ricevuto il messaggio.

Lo **scambio di messaggi non bloccante** consente al mittente di continuare la propria esecuzione dopo la chiamata. La libreria si limita a comunicare al sistema operativo di effettuare la chiamata quando ha tempo di farlo, di conseguenza il mittente non viene bloccato per nulla. Lo svantaggio di questo metodo è che, dopo una `send`, il mittente non può riutilizzare il buffer di messaggi, perché il messaggio precedente potrebbe non essere ancora stato spedito. Deve cercare di sapere in qualche modo se può già riutilizzare il buffer: un'idea è fare *polling* sul sistema, un'altra è prevedere l'invio di un interrupt quando il buffer torna disponibile. Nessuna delle due possibilità semplifica la scrittura del software.

Esaminiamo ora un sistema a scambio di messaggi molto diffuso e disponibile su diversi multicomputer: MPI.

MPI – Interfaccia a scambio di messaggi

Per un certo numero di anni, **PVM** (*Parallel Virtual Machine*) è stato il software di comunicazione più diffuso tra i multicomputer (Geist et al., 1994; Sunderram, 1990), ma di recente è stato rimpiazzato del tutto da **MPI** (*Message-Passing Interface*), molto più articolato e complesso, che fornisce un maggior numero di chiamate di libreria, più opzioni di scelta e molti più parametri per ogni chiamata. La versione originale di MPI, nota come MPI-1, è stata ampliata nel 1997 nella versione MPI-2. Di seguito daremo un'introduzione molto rapida a MPI-1 (che contiene tutti gli elementi basilari), quindi diremo qualcosa circa le aggiunte di MPI-2. Maggiori informazioni sono reperibili su MPI in (Gropp et al., 1994; Snir et al., 1996).

A differenza di PVM, MPI-1 non si occupa della creazione o gestione dei processi e spetta quindi all'utente creare i processi con le chiamate di sistema locali. Una volta creati, i processi sono organizzati in gruppi statici che non possono essere modificati. MPI lavora a livello di questi gruppi.

MPI si basa su quattro concetti principali: comunicatori, tipi di dati dei messaggi, operazioni di comunicazione e topologie virtuali. Un **comunicatore** è un gruppo di processi unito a un contesto. Un contesto è un'etichetta usata per identificare qualcosa, per esempio una fase dell'esecuzione. All'atto della spedizione o della ricezione di messaggi, il contesto può essere usato per evitare che messaggi scorrelati interferiscano gli uni con gli altri.

I messaggi hanno un tipo: ne sono supportati molti, tra cui i caratteri, gli interi corti, regolari o lunghi, i numeri in virgola mobile a precisione singola o doppia, e così via. È anche possibile costruire tipi di dati derivati da questi.

MPI supporta un insieme di operazioni di comunicazione molto vasto. L'operazione fondamentale per spedire un messaggio è la seguente:

```
MPI_Send(buffer, numero, tipo_dati, destinazione, etichetta,
comunicatore)
```

Questa chiamata invia alla destinazione un buffer con `numero` oggetti del tipo di dati specificato. Il campo `etichetta` serve al destinatario perché questi potrebbe specificare di voler ricevere solo messaggi con una certa etichetta. L'ultimo campo specifica il gruppo di processi cui appartiene il destinatario (il campo `destinazione` non è altro che un indice nella lista dei processi del gruppo specificato). La chiamata per la ricezione del messaggio è:

```
MPI_Recv(&buffer, numero, tipo_dati, sorgente, etichetta,
comunicatore, &stato)
```

che specifica il tipo di messaggio atteso dal destinatario, la sorgente di provenienza e l'etichetta.

MPI supporta quattro modalità basilari di comunicazione. La modalità 1 è sincrona: il mittente non può cominciare a spedire finché il destinatario non abbia invocato `MPI_Recv`. La modalità 2 è bufferizzata, per cui la limitazione non esiste più. La modalità 3 è quella standard, cioè dipende dall'implementazione e può essere sincrona o bufferizzata. La modalità 4 è la modalità *pronto*, secondo cui il mittente suppone che il destinatario sia pronto alla comunicazione (come nel caso sincrono) ma non viene effettuato alcun reale controllo. Ciascuna di queste primitive è disponibile in versione bloccante e non bloccante, per un totale di otto primitive. La ricezione ha solo due varianti: bloccante e non bloccante.

MPI fornisce il supporto per la comunicazione collettiva (tra gruppi di processi), comprese le comunicazioni broadcast, *scatter/gather*, a scambio totale, ad aggregazione e a barriera[6].

In tutte le forme di comunicazione collettiva, i processi del gruppo devono effettuare la chiamata e usare argomenti compatibili, altrimenti si verifica un errore. Una forma co-

---

[6] *Scatter* e *gather* sono forme di I/O in cui dati contigui vengono trasferiti in locazioni non contigue (*N.d.Rev.*).

mune di comunicazione collettiva si ha quando i processi sono organizzati ad albero e alcuni valori vengono fatti propagare dalle foglie verso la radice; a ogni passo subiscono un'elaborazione, per esempio la somma di un certo valore o la selezione del valore minimo.

Il quarto concetto alla base di MPI è la **topologia virtuale**, che permette di specificare se i processi sono organizzati ad albero, ad anello, a griglia, in un toro o con un'altra topologia. Questo concetto mette a disposizione un modo per denominare i percorsi di comunicazione e dunque la facilita.

MPI-2 fornisce inoltre i processi dinamici, l'accesso alla memoria distante, la comunicazione collettiva non bloccante, il supporto di I/O scalabile, l'elaborazione in tempo reale e molte altre caratteristiche che vanno oltre lo scopo di questa trattazione. Nella comunità scientifica si è svolta per svariati anni una guerra per la supremazia tra MPI e PVM. I fautori di PVM sostenevano che fosse più semplice da imparare e più facile da usare. I fautori di MPI sostenevano fosse più ricco in dotazione e sottolineavano il fatto che costituiva uno standard formale regolamentato da un documento di definizione ufficiale, a sua volta redatto da parte di una commissione per la standardizzazione. Su questo punto concordavano anche i sostenitori di PVM, ma rispondevano che l'assenza di un apparato burocratico per la piena standardizzazione non costituiva necessariamente un aspetto negativo. Dopo tutto questo batti e ribatti, MPI ha sopravanzato definitivamente PVM nella disputa.

8.4.5 Scheduling

MPI fornisce ai programmatori uno strumento semplice per la creazione di compiti (*job*) il cui svolgimento richiede molteplici CPU e la cui esecuzione impiega un tempo consistente. In presenza di numerose richieste indipendenti da parte di utenti diversi, ciascuna riguardante un certo numero di CPU e lunga un certo lasso di tempo, si rende necessaria la programmazione delle attività del cluster (*scheduling*) per stabilire quale compito deve trovarsi in esecuzione in un determinato momento.

Secondo il modello più semplice, lo scheduler dei compiti richiede che ciascun compito specifichi il numero di CPU di cui ha bisogno. Dopodiché i compiti vengono eseguiti in ordine FIFO, come illustrato nella Figura 8.43(a). Questo modello prevede un controllo, dopo la partenza di un compito, per verificare se sono disponibili abbastanza CPU per avviare il compito successivo nella coda. In caso di esito affermativo il compito può partire, altrimenti il sistema deve aspettare che si liberino altre CPU. Come nota a margine, nella figura abbiamo suggerito la presenza di otto CPU nel cluster, ma potrebbe anche trattarsi di un cluster con 128 CPU allocate in unità di 16 (risultando in otto gruppi di CPU) o di qualche altra combinazione.

Un algoritmo migliore evita il bloccaggio in testa alla coda, perché salta i compiti che sono incompatibili e, scandendo la coda in ordine FIFO, sceglie il primo che è compatibile con le CPU disponibili. Questo algoritmo produce il risultato della Figura 8.43(b).

Un algoritmo di scheduling ancora più sofisticato richiede che ogni compito accodato dichiari la sua forma, ossia il numero di CPU che impegna e il numero di minuti che impiega per l'esecuzione. Grazie a questa informazione, lo scheduler dei compiti può cercare una configurazione che riempia al meglio il rettangolo numero di CPU-tempo. Questa operazione si dice *tiling* ("rivestimento con piastrelle") ed è molto efficace in quelle situazioni in cui le richieste dei compiti pervengono durante il giorno e la loro esecuzione è prevista durante la notte;

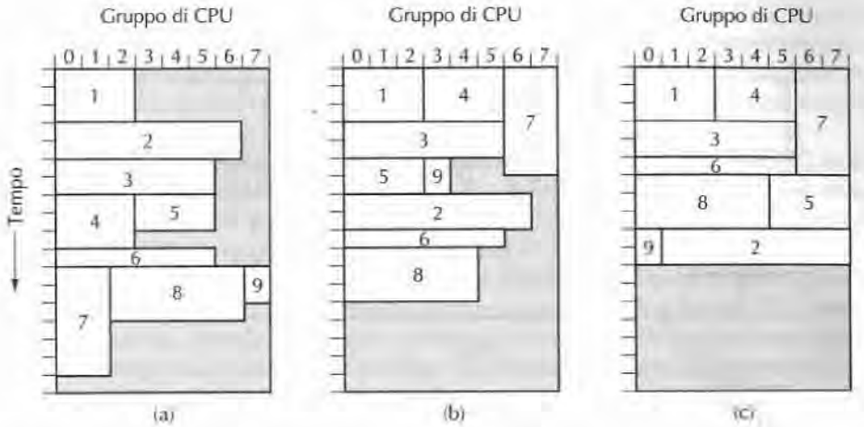


Figura 8.43 Scheduling di un cluster. (a) FIFO. (b) Senza bloccaggio in testa alla coda. (c) Tiling. Le parti ombreggiate indicano le CPU inattive.

in questo modo lo scheduler dei compiti ottiene tutte le informazioni in anticipo e può far eseguire i compiti secondo l'ordine ottimale, come mostrato nella Figura 8.43(c).

8.4.6 Memoria condivisa a livello applicativo

Gli esempi proposti dovrebbero aver dimostrato come i multicomputer raggiungano dimensioni molto maggiori rispetto ai multiprocessori. Questo fatto ha portato allo sviluppo di sistemi a scambio di messaggi come MPI, eppure molti programmatori non amano questo modello e vorrebbero poter lavorare con l'illusione di una memoria condivisa, anche se non esiste davvero. Riuscire a soddisfare questa richiesta vorrebbe dire mettere insieme il meglio di entrambi i paradigmi: hardware economico in quantità enormi, e programmazione semplice. Questo obiettivo è il Santo Graal del calcolo parallelo.

Molti ricercatori sono giunti alla conclusione che la condivisione di memoria non funziona bene su sistemi molto grandi se implementata a livello di architettura, ma che sono possibili altri modi per raggiungere lo stesso scopo. Già dalla Figura 8.19 sappiamo che la memoria condivisa può essere introdotta ad altri livelli, nei paragrafi seguenti vedremo com'è possibile introdurre la memoria condivisa nel modello di programmazione di un multicomputer, benché non esista a livello hardware.

Memoria condivisa distribuita

Un tipo di sistema a memoria condivisa a livello applicativo è quello dei sistemi basati su pagine, chiamati spesso **DSM** (*Distributed Shared Memory*). L'idea è semplice: un insieme di CPU di un multicomputer condivide uno spazio di indirizzi virtuali paginato. Nella versione più semplice, ogni pagina è contenuta nella RAM di una sola CPU. La Figura 8.44(a) mostra uno spazio di indirizzi virtuali condiviso e formato da 16 pagine, disseminate tra quattro CPU.

Quando una CPU fa riferimento a una pagina che si trova nella sua RAM locale, la lettura o scrittura si svolge senza alcun ritardo, ma il riferimento a una pagina in una memoria

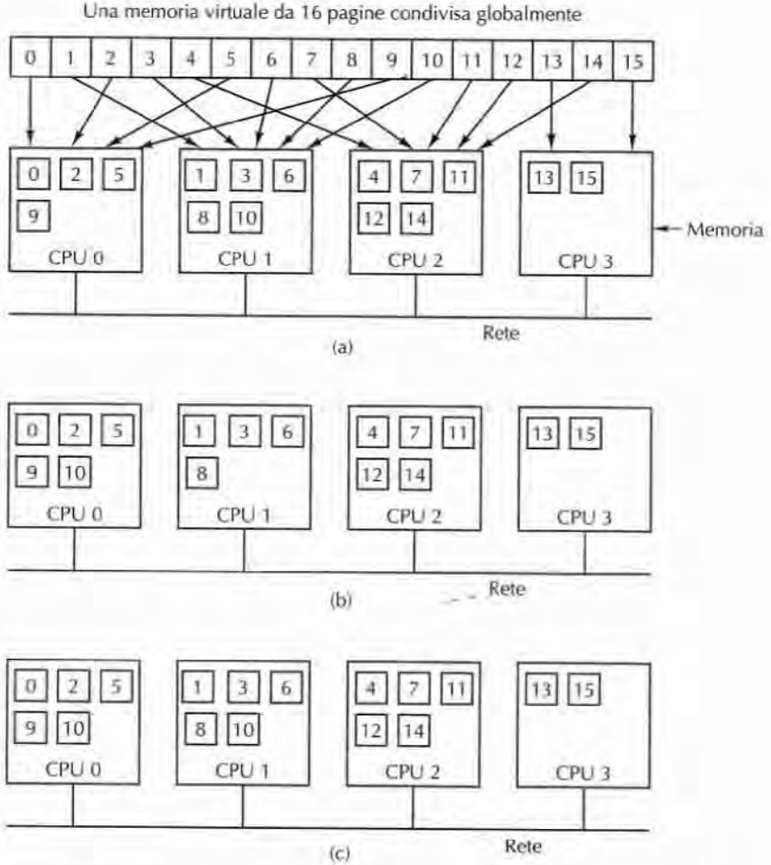


Figura 8.44 Spazio degli indirizzi virtuali con 16 pagine distribuite su quattro nodi di un multicomputer. (a) Situazione iniziale. (b) Dopo un riferimento alla pagina 10 da parte della CPU 0. (c) Dopo un riferimento alla pagina 10 da parte della CPU 1 (nell'ipotesi di una pagina di sola lettura).

distante provoca un errore di pagina. La differenza qui è che la pagina mancante non viene caricata dal disco, bensì il sistema *runtime* o il sistema operativo spediscono un messaggio al nodo che la detiene affinché se ne privi e la spedisca al richiedente. Una volta giunta a destinazione, la pagina viene mappata nella RAM locale e l'istruzione che ha provocato l'errore può essere eseguita, proprio come succede dopo un normale errore di pagina. La Figura 8.44(b) illustra la situazione dopo che la CPU 0 ha causato un errore sulla pagina 10, che viene trasferita conseguentemente dalla CPU 1 alla CPU 0.

Questa semplice idea, implementata per la prima volta in IVY (Li e Hudak, 1986, 1989), consente l'utilizzo in un multicomputer di una memoria pienamente condivisa a consistenza sequenziale, anche se sono possibili ulteriori ottimizzazioni per migliorarne le prestazioni.

La prima variante di IVY ammette che le pagine contrassegnate in sola lettura siano presenti simultaneamente in più nodi. Perciò quando si verifica un errore di pagina, viene sì spedita una copia della pagina alla macchina richiedente, ma la versione originale resta al suo posto perché non c'è pericolo di conflitto. La Figura 8.44(c) mostra la situazione in cui due CPU condividono una pagina in sola lettura (la pagina 10).

Nonostante questa ottimizzazione, le prestazioni sono spesso inaccettabili, specie nei casi in cui un processo scrive qualche parola all'inizio di una pagina mentre un altro processo sta scrivendo alcune parole in fondo alla stessa pagina. L'esistenza di una sola copia della pagina costringe la sua spedizione ripetuta avanti e indietro, una condizione nota come **falsa condivisione**. Questo problema può essere affrontato in modi diversi. Nel sistema Treadmarks, per esempio, si abbandona la consistenza sequenziale a favore della consistenza dopo rilascio (Amza, 1996). Una pagina potenzialmente scrivibile può trovarsi contemporaneamente presso più nodi, ma un processo che voglia effettuare una scrittura al suo interno deve prima segnalare le proprie intenzioni eseguendo un'operazione di `acquire`. A questo punto vengono invalidate tutte le copie della pagine tranne la più recente e non è possibile effettuare nuove copie finché non viene eseguita la `release` corrispondente; solo allora la pagina può tornare a essere condivisa.

Una seconda ottimizzazione implementata in Treadmarks è la dichiarazione iniziale in modalità di sola lettura delle pagine scrivibili. All'atto della prima scrittura in una pagina, viene sollevato l'errore di protezione e il sistema crea una copia della pagina chiamata **pagina gemella**. Quindi la pagina originale viene classificata con modalità di lettura/scrittura e le scritture successive possono procedere appieno ritmo. Se più tardi si verifica un errore di pagina e bisogna spedire la pagina al richiedente, viene effettuato un confronto parola per parola tra la pagina corrente e la sua gemella e vengono spedite solo le parole che sono effettivamente cambiate, riducendo così la dimensione del messaggio.

Il verificarsi di un errore di pagina pone il problema di localizzare la pagina mancante. Sono possibili svariate soluzioni, incluse quelle adottate nelle macchine NUMA e COMA, come le directory (basate sui nodi di appartenenza). Infatti, molte soluzioni usate nel contesto DSM sono applicabili anche ai contesti NUMA e COMA, perché DSM non è altro che un'implementazione software di NUMA o COMA, in cui ogni pagina è trattata come una linea di cache.

I sistemi DSM sono oggetto di intensa ricerca. Tra i sistemi più interessanti annoveriamo CASHMERE (Kontothanassis et al., 1997; Stets et al., 1997), CRL (Johnson et al., 1995), Shasta (Scales et al., 1996) e Treadmarks (Amza, 1996; Lu et al., 1997).

Linda

I sistemi DSM basati su pagine come IVY e Treadmarks usano l'hardware della MMU per intercettare gli accessi alle pagine mancanti. Benché la spedizione delle differenze tra pagine migliori le prestazioni rispetto all'invio delle pagine intere, rimane il fatto che le pagine sono unità inadatte alla condivisione. Per questo motivo sono stati tentati approcci alternativi al problema.

Linda, uno di questi tentativi, fornisce la gestione di processi su macchine diverse con il supporto di una memoria condivisa distribuita altamente strutturata (Carriero e Gelernter, 1989). L'accesso alla memoria avviene tramite un piccolo insieme di operazioni primitive che possono essere aggiunte ai linguaggi esistenti (C o FORTRAN) per costituire linguaggi paralleli, in questo caso C-Linda e FORTRAN-Linda.

("abc"; 2; 5)
("matrice"; 1; 6; 3,14)
("famiglia"; "è sorella"; Garolina; Eleonora)

Figura 8.45 Tre tuple di Linda.

Il concetto unificatore di Linda è l'esistenza di uno **spazio di tuple** astratto, globale al sistema e accessibile da ogni suo processo. Lo spazio di tuple è simile a una memoria globale condivisa, dotata in più di una struttura intrinseca. Lo spazio delle tuple contiene un certo numero di **tuple**, ciascuna costituita a sua volta da uno o più campi. Nel caso di C-Linda, i tipi dei campi comprendono gli interi, gli interi lunghi e i numeri in virgola mobile, oltre ai tipi composti quali gli array (comprese le stringhe) e le strutture (ma non altre tuple). La Figura 8.45 mostra tre esempi di tuple.

Sono disponibili quattro operazioni sulle tuple. La prima, `out`, inserisce una tupla nello spazio delle tuple. Per esempio

```
out("abc"; 2; 5);
```

inserisce la tupla ("abc", 2, 5) nello spazio delle tuple. I campi di `out` sono in genere costanti, variabili o espressioni, come in

```
out("matrice"; i; j; 3,14);
```

che produce una tupla con quattro campi, dove il secondo e il terzo sono specificati dal valore corrente delle variabili *i* e *j*.

Le tuple vengono prelevate dallo spazio delle tuple tramite la primitiva `in`; il loro indirizzamento è per contenuto e non per nome o per indirizzo. I campi di `in` possono essere espressioni o parametri formali. Si consideri, per esempio

```
in("abc"; 2; ? i);
```

Questa operazione "cerca" nello spazio delle tuple una tupla formata dalla stringa "abc", dall'intero 2 e da un terzo campo che contiene un intero qualsiasi (ipotizziamo *i* sia di tipo intero). Se viene trovata, la tupla è rimossa dallo spazio delle tuple e viene assegnato alla variabile *i* il valore del terzo campo della tupla. Le operazioni di ritrovamento e rimozione sono atomiche, perciò se due processi eseguono simultaneamente la stessa operazione `in`, solo uno di loro avrà successo, a meno che non esistano due o più tuple corrispondenti ai criteri di ricerca. Lo spazio delle tuple potrebbe contenere addirittura copie multiple della stessa tupla.

L'algoritmo di ricerca usato da `in` è semplice. Concettualmente si procede così: i campi della primitiva `in`, che ne costituiscono il **template** (cioè lo schema, il modello), vengono confrontati con i campi corrispondenti di tutte le tuple dello spazio. Si verifica una corrispondenza se risultano soddisfatte le tre condizioni:

1. il template e le tuple hanno lo stesso numero di campi
2. i tipi dei campi corrispondenti sono uguali
3. ogni costante o variabile del template si accorda al rispettivo campo della tupla.

I parametri formali, indicati dal punto interrogativo seguito da un nome di variabile o da un tipo, non partecipano al confronto (fatta eccezione per quello di tipo); ciononostante, quelli che contengono una variabile vengono assegnati al termine di una ricerca fruttuosa.

Se non si trova alcuna tupla corrispondente ai criteri, il processo chiamante resta sospeso finché un altro processo non inserisca la tupla richiesta, al che il primo viene automaticamente risvegliato e gli viene consegnata la nuova tupla. Il bloccaggio e lo sbloccaggio automatico dei processi garantisce che, se un processo sta per produrre una tupla e un altro sta per prelevarla, non importa chi dei due comincia per primo.

Oltre a `in` e `out`, Linda fornisce anche la primitiva `read` che opera analogamente a `in`, ma non rimuove la tupla dallo spazio. Esiste anche la primitiva `eval` che richiede la valutazione parallela dei suoi parametri e l'inserzione della tupla risultante nello spazio delle tuple. Questo meccanismo permette di svolgere qualsiasi tipo di calcolo ed è il modo in cui sono creati i processi paralleli in Linda.

Un paradigma di programmazione tipico di Linda è il **replicated worker model** ("modello del lavoratore replicato"). Questo modello si basa sull'idea di una **task bag** ("borsa dei lavori") piena di compiti da svolgere. Il processo principale inizia la sua esecuzione con un ciclo che contiene

```
out("task-bag", compito);
```

che a ogni iterazione inserisce nello spazio delle tuple la descrizione di un compito diverso. Ogni lavoratore comincia con il prelevare una tupla di descrizione di un compito con

```
in("task-bag", ?compito);
```

e poi inizia a svolgerlo; al suo compimento ne preleva un altro e così via. È anche possibile che vengano inseriti nella task bag nuovi lavori durante l'esecuzione. Così facendo, il lavoro viene suddiviso dinamicamente tra i lavoratori e ciascuno di loro resta costantemente impegnato; il tutto funziona con una produzione e uno scambio d'informazioni accessorie relativamente contenuti.

Esistono varie implementazioni di Linda per sistemi multicomputer e tutte si trovano ad affrontare la questione chiave della distribuzione delle tuple tra le macchine e del modo di localizzarle quando richieste. Tra le diverse possibilità ci sono la tecnica broadcast e le directory. Anche la duplicazione è una questione critica da affrontare; per un'analisi di tutti questi aspetti si veda (Bjornson, 1993).

Orca

Un approccio un po' diverso per la condivisione di memoria a livello applicativo nei multicomputer è l'uso di oggetti veri e propri come unità della condivisione invece di semplici tuple. Gli oggetti sono costituiti da uno stato interno (nascosto) unito ai metodi per agire su quello stato. La scelta di non permettere ai programmatori di accedere direttamente allo stato degli oggetti offre molte modalità di condivisione tra macchine che non hanno una memoria fisica condivisa.

Un sistema basato su oggetti che fornisce l'illusione di una memoria condivisa nei multicomputer è Orca (Bal, 1991; Bal et al., 1992; Bal e Tanenbaum, 1988), un linguaggio di programmazione tradizionale (basato su Modula 2) con l'aggiunta di due nuove caratteristiche: gli oggetti e la capacità di creare nuovi processi. Un oggetto di Orca è un tipo di dati astratto, analogo a un oggetto di Java o a un package di Ada, che incapsula le strutture dati interne e i metodi scritti dall'utente, chiamati **operazioni**. Gli oggetti sono passivi,

```
Object implementation stack;
  top:integer;                          # allocazione dello stack e puntatore alla cima
  stack: array [integer 0..N-1] of integer;

  operation push(item: integer);        # funzione che non restituisce valori
  begin
    guard top < N - 1 do
      stack[top] := item;               # push di un elemento sullo stack
      top := top + 1;                   # incremento del puntatore allo stack
    od;
  end;

  operation pop( ): integer;            # funzione che restituisce un intero
  begin
    guard top > 0 do                    # sospensione in caso di stack vuoto
      top := top - 1;                   # decremento del puntatore allo stack
      return stack[top];                # restituzione del valore in cima allo stack
    od;
  end;

begin
  top := 0;                             # inizializzazione
end;
```

Figura 8.46 Versione semplificata di un oggetto stack di Orca comprendente i dati interni e due operazioni.

cioè non contengono thread cui inviare messaggi, mentre i processi sono attivi e accedono ai dati interni di un oggetto invocando i suoi metodi.

Ogni metodo di Orca consiste in una lista di coppie (sentinella, blocco d'istruzioni). La sentinella è un'espressione booleana la cui valutazione non ha effetti collaterali, oppure una sentinella vuota che assume sempre il valore *true*. All'invocazione di un'operazione vengono valutate tutte le sue sentinelle in un ordine qualsiasi: se valgono tutte *false*, allora il processo chiamante viene sospeso finché almeno una non diventi *true*. Quando si trova almeno una sentinella con valore *true*, il blocco d'istruzioni corrispondente viene eseguito. La Figura 8.46 rappresenta un oggetto *stack* con due operazioni, *push* e *pop*.

Una volta definito uno *stack*, è possibile dichiarare variabili di questo tipo con

```
s, t: stack;
```

che crea due oggetti stack e inizializza la variabile *top* di ciascuna al valore 0. La variabile intera *k* può essere impilata sullo stack *s* tramite l'istruzione

```
s$push(k);
```

e così via. L'operazione *pop* ha una sentinella, perciò un tentativo di estrarre una variabile dalla cima di uno stack vuoto causerà la sospensione del chiamante finché un altro processo non impilerà qualcosa sullo stack.

Orca definisce un'istruzione **fork** per creare nuovi processi all'interno di un processore specificato dall'utente. Il nuovo processo esegue la procedura indicata nell'istruzione `fork`. È possibile passare al nuovo processo alcuni parametri, inclusi gli oggetti, ed è questo il modo in cui gli oggetti diventano distribuiti tra macchine. Per esempio l'istruzione

for i **in** 1 .. n **do fork** foobar(s) **on** i; **od;**

genera un nuovo processo su tutte le macchine dalla 1 alla *n*, e ciascuno di loro esegue il programma *foobar* all'interno della macchina corrispondente. Poiché gli *n* nuovi processi (e il loro genitore) girano in parallelo, possono tutti eseguire *push* e *pop* di elementi sullo stack condiviso *s* come se si trovassero in un multiprocessore a memoria condivisa. Il sistema *runtime* crea l'illusione di una memoria condivisa, che in realtà non esiste.

Le operazioni sugli oggetti condivisi sono atomiche e sequenzialmente consistenti. Il sistema garantisce che, se diversi processi eseguono operazioni sullo stesso oggetto condiviso quasi in simultanea, allora sceglierà di eseguirle in un certo ordine e tutti i processi osserveranno lo stesso ordine di eventi.

Orca integra la gestione dei dati condivisi e della sincronizzazione in un modo non presente nei sistemi DSM basati su pagine. I programmi paralleli hanno bisogno di due tipi di sincronizzazione: il primo è la sincronizzazione con mutua esclusione, che impedisce a due processi di trovarsi contemporaneamente all'interno di una regione critica. In Orca, ogni operazione su di un oggetto condiviso è in tutto simile a una regione critica, perché il sistema garantisce che il risultato finale di un insieme di operazioni parallele è lo stesso risultato che si otterrebbe dopo una loro esecuzione sequenziale. Da questo punto di vista, un oggetto di Orca è simile a una forma distribuita di monitor (Hoare, 1975).

L'altro tipo di sincronizzazione è quella su condizione, in cui un processo si blocca in attesa del soddisfacimento di una certa condizione. In Orca, la sincronizzazione su condizione si implementa con le sentinelle. Nell'esempio della Figura 8.46, se un processo cerca di eseguire una *pop* di un elemento da uno stack vuoto, resterà sospeso fin quando lo stack non sarà più vuoto.

Il sistema runtime di Orca gestisce la duplicazione, migrazione e coerenza di oggetti, e l'invocazione delle loro operazioni. Ogni oggetto può trovarsi in uno di due stati: essere una copia unica o un duplicato. Un oggetto nello stato di copia unica esiste in una sola macchina, perciò tutte le richieste che lo riguardano sono spedite lì. Un oggetto replica è presente su tutte le macchine che contengono un processo che lo usa; ciò semplifica le operazioni di lettura (che possono essere svolte localmente), ma accresce il costo delle operazioni di aggiornamento. Quando si vuol eseguire un'operazione che modifica un oggetto duplicato, per prima cosa l'operazione deve procurarsi un numero di sequenza presso un apposito processo centralizzato. Dopodiché viene spedito un messaggio a tutte le macchine che detengono una copia dell'oggetto, ordinando loro di eseguire l'operazione. Poiché tutte le operazioni di aggiornamento sono associate a un numero sequenziale, alle macchine non resta altro che eseguire le operazioni in quell'ordine, garantendo così la consistenza sequenziale.

Globe

Molti DSM, così come anche Linda e Orca, girano su sistemi localizzati, ovvero compresi all'interno di un unico edificio o campus. D'altra parte, è anche possibile costruire sistemi multicomputer, con memoria condivisa a livello applicativo, che lavorino su scala globale. Il sistema Globe consente di localizzare gli oggetti in uno spazio degli indirizzi condiviso da svariati processi che possono trovarsi in esecuzione su macchine residenti in continenti diversi (Kermarrec et al., 1998; Popescu et al., 2002; Van Steen et al., 1999). Per accedere ai dati di un oggetto condiviso, un processo utente deve usare i propri metodi, il che permette di differenziare le strategie implementative a seconda del tipo di oggetto. Per esempio, si può mantenere una copia unica dei dati da far circolare dinamicamente su richiesta (una buona

soluzione se i dati sono aggiornati frequentemente da un unico processo proprietario), oppure replicare i dati in tutte le copie dell'oggetto e inviare gli aggiornamenti a ogni copia tramite un protocollo di trasmissione uno-a-molti affidabile.

Globe ha l'ambizione di raggiungere un miliardo di utenti e mille miliardi di oggetti che possano migrare nel sistema. La localizzazione degli oggetti e la loro gestione diventano cruciali al crescere delle dimensioni del sistema stesso. La soluzione di Globe consiste nel fornire uno schema generale in cui ciascun oggetto può specificare le proprie strategie di duplicazione, di sicurezza, e così via. Con ciò si superano i problemi presenti in altri sistemi e dovuti alla pretesa di adottare una strategia unica per tutte le situazioni, e si preserva al contempo la facilità della programmazione offerta dalla condivisione di memoria.

Tra gli altri sistemi distribuiti su aree di vaste dimensioni ricordiamo Globus (Foster e Kesselman, 1998a; Foster e Kesselman, 1998b) e Legion (Grimshaw e Wulf, 1996; Grimshaw e Wulf, 1997), che però non supportano lo stesso grado di condivisione di memoria garantito da Globe.

8.4.7 Prestazioni

Lo scopo della costruzione di un calcolatore parallelo è di renderlo più veloce di una macchina monoprocessore; se non raggiunge questo semplice obiettivo, tanto vale farne a meno. Inoltre, l'obiettivo dovrebbe essere soddisfatto in modo conveniente dal punto di vista dei costi: un sistema due volte più veloce di una macchina uniprocessore, ma che costa 50 volte tanto, non è un grande affare. In questo paragrafo ci occupiamo delle questioni legate alle prestazioni che riguardano le architetture per il calcolo parallelo.

Parametri di valutazione hardware

In una prospettiva hardware, i parametri significativi per la misura delle prestazioni sono la velocità di CPU e I/O, e le prestazioni della rete d'interconnessione. La velocità di CPU e I/O si valuta come nel caso uniprocessore, mentre il parametro chiave che discrimina il caso parallelo è associato all'interconnessione. Esistono due concetti chiave, la latenza e la larghezza di banda, che presentiamo in successione.

La latenza di andata e ritorno (*roundtrip latency*) è il tempo che trascorre da quando una CPU spedisce un pacchetto a quando ne riceve una risposta. Se il pacchetto viene spedito alla memoria, la latenza misura il tempo di lettura o scrittura di un blocco di parole, se invece è destinato a un'altra CPU, misura il tempo di comunicazione tra processori per pacchetti di quella dimensione. In genere si è interessati alla latenza dei pacchetti di dimensione minima, spesso una parola o una piccola linea di cache.

La latenza è influenzata da svariati fattori e cambia in base al tipo d'instradamento: a commutazione di circuito, store-and-forward, a cut through virtuale (o *wormhole*). Nel primo caso, la latenza è la somma del tempo di configurazione e del tempo di trasmissione. La configurazione di un circuito avviene tramite la spedizione di un pacchetto sonda che serve a riservare le risorse e a riferire gli esiti dell'operazione al suo ritorno. Dopo di ciò, bisogna assemblare il pacchetto dati e, una volta pronto, è possibile spedire i bit a piena velocità. Perciò, se il tempo totale di configurazione è T_s, se il pacchetto contiene p bit e la larghezza di banda è di b bit/s, la latenza di sola andata è di $T_s + p/b$ secondi. Se il circuito è

full duplex, il ritorno non comporta alcun tempo di configurazione, e così la latenza minima per la spedizione di un pacchetto di p bit, e per la ricezione di p bit di risposta, è di $T_s + 2p/b$ secondi.

Nel caso della commutazione di pacchetto, non è necessario spedire in anticipo un pacchetto sonda alla destinazione, ma ci vuole comunque un certo tempo di configurazione T_a per assemblare il pacchetto. Il tempo di trasmissione di sola andata è $T_a + p/b$, ma si tratta solo del tempo impiegato per raggiungere il primo commutatore. All'interno del commutatore la trasmissione subisce un ritardo finito, chiamiamolo T_d, e il procedimento si ripete nei commutatori successivi. Il ritardo T_d comprende sia il tempo di elaborazione, sia il ritardo dovuto all'attesa nella coda perché si liberi una porta disponibile alla spedizione. Se ci sono n commutatori, allora la latenza totale di sola andata è data dalla formula $T_a + n(p/b + T_d) + p/b$, dove l'ultimo termine è richiesto dalla copia tra l'ultimo commutatore e la destinazione.

Nel caso dell'instradamento a cut through virtuale, la latenza di sola andata è nel migliore dei casi $T_a + p/b$ secondi, poiché non ci sono pacchetti sonda da inviare per configurare il circuito, né ritardi dovuti all'operazione di store-and-forward. In sostanza è richiesto solo il tempo di configurazione iniziale per assemblare il pacchetto, più il tempo necessario alla spedizione dei bit. In entrambi i casi va poi aggiunto il ritardo di propagazione, che spesso è trascurabile.

L'altra metrica hardware è la larghezza di banda. Molti programmi paralleli spostano grandi quantità di dati, perciò il numero di byte trasferibili in ogni secondo diventa un parametro cruciale per le loro prestazioni. Esistono molte metriche per la misurazione della larghezza di banda e ne abbiamo già incontrata una: la larghezza di banda di bisezione. Un'altra è la **larghezza di banda complessiva** (*aggregate bandwidth*), definita come la somma delle capacità di tutti i collegamenti. Questa quantità indica il numero massimo di bit che può transitare simultaneamente sulla rete. Un altro parametro importante è la larghezza di banda media in uscita da ogni CPU. Se ciascuna CPU è in grado di spedire dati a 1 MB/s, il sistema non trae alcun giovamento dall'avere una larghezza di banda di bisezione di 100 GB/s. La comunicazione sarà comunque limitata dal quantitativo di dati che può essere emesso da ciascuna CPU.

Nella pratica è pressoché impossibile avvicinarsi ai limiti teorici della larghezza di banda perché bisogna svolgere molte attività accessorie che riducono la capacità. Per esempio, ogni pacchetto richiede sempre lo svolgimento di alcune attività preparatorie come l'assemblaggio, la costruzione della sua intestazione e la spedizione. Spedire 4 pacchetti da 1024 byte non potrà mai raggiungere la stessa larghezza di banda della spedizione di un pacchetto da 4096 byte. Sfortunatamente però, la spedizione di pacchetti piccoli aiuta a diminuire la latenza, dal momento che i pacchetti grandi occupano linee e commutatori per troppo tempo. C'è dunque un'incompatibilità insita nel cercare di raggiungere latenze medie contenute congiuntamente a larghezze di banda elevate. Le prime sono importanti per alcune applicazioni, in altri casi è vero l'esatto contrario. Vale la pena notare che, comunque, è sempre possibile procurarsi maggiore larghezza di banda (aggiungendo cavi su cavi), ma che non è possibile comprare latenze più basse. Perciò è buona regola eccedere nel tentativo di ridurre la latenza il più possibile, per poi preoccuparsi in un secondo momento della larghezza di banda.

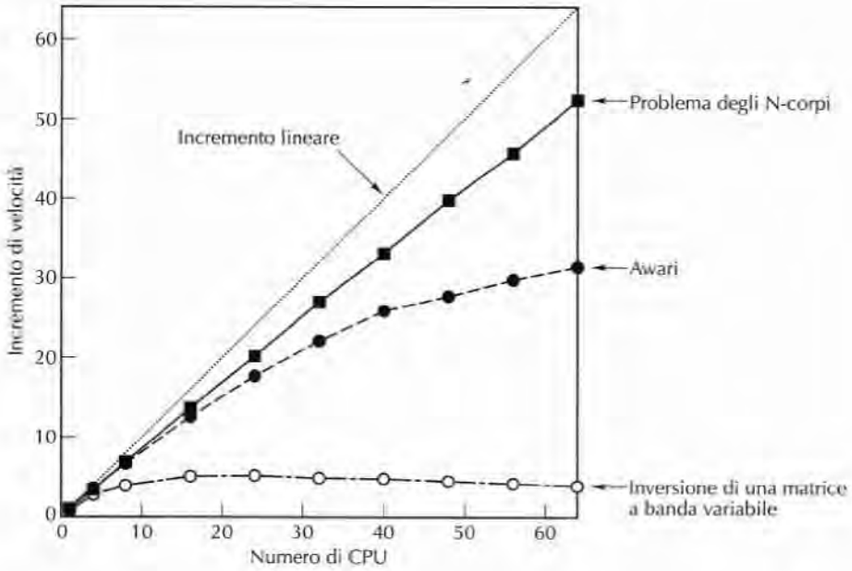


Figura 8.47 I programmi reali hanno incrementi di velocità minori di quello ideale (linea punteggiata).

Parametri di valutazione software

Le metriche appena illustrate, quali la latenza e la larghezza di banda, si riferiscono alle capacità dell'hardware, ma l'interesse degli utenti è rivolto verso una prospettiva diversa: vogliono sapere quanto si ridurranno i tempi di esecuzione passando da un sistema monoprocessore a uno parallelo. Per quanto li riguarda, la metrica fondamentale è il fattore d'incremento della velocità, cioè di quante volte risulta più veloce l' esecuzione di un programma su un sistema a n processori rispetto a un sistema a monoprocessore. L'andamento di questo rapporto viene presentato spesso mediante grafici simili a quello della Figura 8.47, in cui sono raffigurati diversi programmi paralleli eseguiti su di un multicomputer costituito da 64 CPU Pentium Pro. Ogni curva mostra l'incremento di velocità di un programma eseguito su k CPU, in funzione di k. La linea punteggiata indica l'incremento massimo possibile che corrisponde alla situazione in cui l'uso di k CPU accelera k volte il programma, per qualsiasi valore di k. Non sono molti i programmi che raggiungono l'incremento ideale, ma alcuni ci vanno vicini. La versione parallela della soluzione al problema degli N-corpi funziona molto bene[7]. La simulazione di awari (un gioco da tavolo di origine africana che si giocava in buchette scavate per terra) si comporta ragionevolmente bene. Al contrario l'inversione di una matrice a banda variabile (*skyline matrix*) non supera mai più di cinque volte la velocità di esecuzione su macchine monoprocessore, indipendentemente dal numero

[7] Si tratta del classico problema dell'evoluzione dinamica di un sistema di N masse gravitazionali (*N.d.Rev.*).

di processori disponibili. Per un excursus su questi programmi e sui relativi risultati si veda (Bal et al., 1998).

L'impossibilità pratica di raggiungere l'incremento ideale è dovuta in parte al fatto che quasi tutti i programmi hanno qualche componente sequenziale, di solito la fase di inizializzazione, la lettura dei dati o la raccolta dei risultati. In queste situazioni la disponibilità di più CPU non aiuta affatto. Nella Figura 8.48(a) mostriamo un programma che gira in T secondi su un monoprocessore, laddove una frazione f di questo tempo è codice sequenziale e la rimanente frazione $(1 - f)$ è potenzialmente parallelizzabile. Se questa seconda parte di codice può essere distribuita tra n CPU senza subire rallentamenti, allora il suo tempo di esecuzione può ridursi al meglio da $(1 - f)T$ a $(1 - f)T/n$ secondi, come indicato nella Figura 8.48(b). In ragione di ciò, il tempo totale di esecuzione risulta di $fT + (1 - f)T/n$ secondi. Il fattore d'incremento è esattamente il rapporto tra il tempo di esecuzione del programma originale, T, e il nuovo tempo di esecuzione:

$$\textit{Incremento di velocità} = \frac{n}{1 + (n-1)f}$$

Quando $f = 0$ otteniamo l'incremento lineare, ma per $f > 0$ l'incremento perfetto diventa impossibile a causa della componente sequenziale. Questa formula è nota come **legge di Amdahl**.

Questa legge non è la sola ragione per l'impossibilità del raggiungimento dell'incremento ideale. Un ruolo in tal senso è giocato anche dalle latenze di comunicazione, dalle limitazioni poste dalla larghezza di banda e dalle inefficienze algoritmiche. Per di più, anche disponendo di 1000 CPU, non tutti i programmi possono essere scritti per utilizzare un numero di processori così elevato, e il tempo necessario per avviarli tutti potrebbe risultare significativo. Va aggiunto che, alle volte, l'algoritmo migliore per un determinato problema non è parallelizzabile efficacemente e quindi bisogna accontentarsi di un algoritmo parallelo subottimale. Al di là di queste osservazioni, è evidente che per molte applicazioni può far comodo che un programma giri n volte più velocemente, anche se ciò comporta l'uso di $2n$ CPU. Dopo tutto le CPU non sono tanto costose e molte aziende proliferano pur raggiungendo percentuali di efficienza considerevolmente inferiori al 100%.

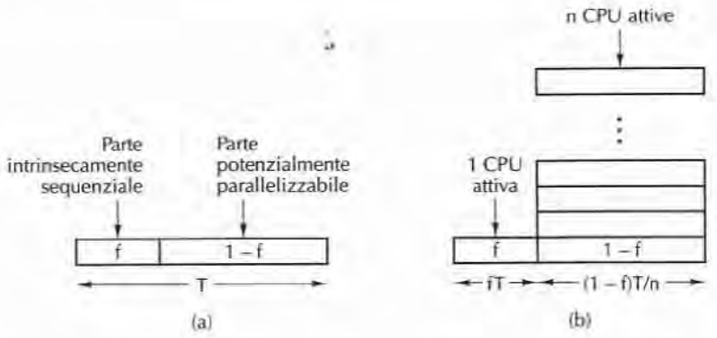


Figura 8.48 (a) I programmi hanno una parte sequenziale e una parallelizzabile. (b) Effetto dell'esecuzione parallela di una parte del programma.

Miglioramento delle prestazioni

Il modo più diretto per migliorare le prestazioni è aggiungere nuove CPU al sistema, benché questa aggiunta vada fatta in modo da non creare colli di bottiglia. I sistemi in cui la potenza di calcolo aumenta proporzionalmente al numero di CPU aggiunte si dicono **scalabili**.

Per meglio comprendere questo concetto, consideriamo il sistema della Figura 8.49(a) composto da quattro CPU collegate da un bus. Scalando il sistema alla dimensione 16, aggiungendo cioè 12 nuove CPU, ci riportiamo alla situazione della Figura 8.49(b). Se il bus ha larghezza di banda pari a b MB/s, la quadruplicazione del numero di CPU riduce la larghezza di banda disponibile per ogni CPU a $b/16$ MB/s. Un sistema siffatto non è scalabile.

Ripetiamo ora la stessa operazione con il sistema a griglia delle Figure 8.49(c)-(d). In questa topologia, l'aggiunta di nuove CPU comporta la creazione di nuovi collegamenti, perciò la crescita delle dimensioni del sistema non provoca la diminuzione della larghezza di banda per CPU che abbiamo osservato nel sistema basato su bus. Infatti, il rapporto tra numero di collegamenti e numero di CPU aumenta da 1 (4 CPU, 4 collegamenti) a 1,5 (16 CPU, 24 collegamenti), dunque l'aggiunta di nuove CPU giova alla larghezza di banda per CPU.

Naturalmente la larghezza di banda non è il solo parametro significativo. Il collegamento al bus di nuove CPU non aumenta il diametro della rete d'interconnessione, né la latenza in assenza di traffico, ma non è così per la griglia. Il diametro di una griglia $n \times n$ è lungo $2(n - 1)$, perciò la latenza nel caso medio e peggiore aumenta grosso modo come la radice quadrata del numero di CPU. Un sistema con 400 CPU ha diametro 38, uno con 1600 CPU ha diametro 78, dunque quadruplicare il numero di CPU equivale approssimativamente a raddoppiare il diametro e quindi la latenza media.

Un sistema scalabile dovrebbe mantenere idealmente la stessa larghezza di banda media per CPU e la stessa latenza media al crescere delle sue dimensioni. Tuttavia nella pratica della progettazione, se si riesce spesso a garantire sufficiente larghezza di banda per CPU, in genere si osserva però un aumento della latenza al crescere delle dimensioni del progetto. Una crescita logaritmica, come quella garantita da un ipercubo, è quanto di meglio si possa fare.

La latenza è un parametro cruciale per le prestazioni di molte applicazioni a granularità fine e media (la granularità di un'applicazione si definisce come il rapporto tra il tempo di calcolo e il tempo di comunicazione). Se un programma ha bisogno di dati che non si trovano nella sua memoria locale, allora sperimenterà un certo ritardo nel procurarseli che sarà tanto maggiore quanto più grande è il sistema, come abbiamo appena argomentato. Il problema si presenta ugualmente nei multiprocessori e nei multicomputer, perché in entrambi i casi la memoria fisica si trova divisa in moduli molto distanti gli uni dagli altri.

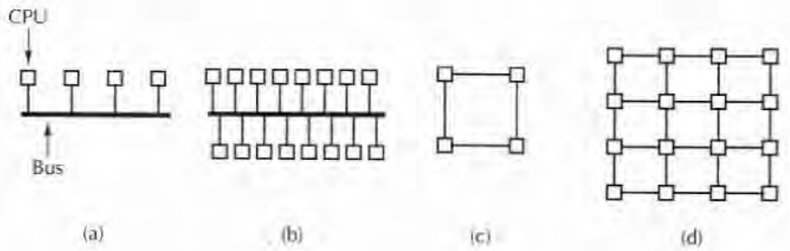


Figura 8.49 (a) Un sistema a bus con 4 CPU. (b) Un sistema a bus con 16 CPU. (c) Un sistema a griglia con 4 CPU. (d) Un sistema a griglia con 16 CPU.

In conseguenza di ciò, molti progettisti di sistemi cercano di intervenire su queste distanze per ridurre la latenza o almeno per nasconderla, avvalendosi di diverse tecniche che ci apprestiamo a menzionare. La prima tecnica per mascherare la latenza è la duplicazione dei dati: se presso diverse locazioni esistono delle copie di un blocco di dati, gli accessi provenienti da queste locazioni possono trarne giovamento. Una tecnica di questo tipo è il caching, secondo cui una o più copie dei dati sono conservate in prossimità delle locazioni che li utilizzano o cui "appartengono". D'altro canto, un'altra strategia prevede di mantenere copie di pari grado, cioè che hanno lo stesso stato, invece di utilizzare la relazione gerarchica e asimmetrica del caching che distingue tra copia primaria e secondaria. Quando si gestiscono le duplicazioni, implementate sotto qualsiasi forma, bisogna affrontare i problemi relativi alla localizzazione dei blocchi di dati, alla tempistica dei trasferimenti e all'identificazione dei loro detentori. Le risposte a questi problemi vanno dall'allocazione dinamica su richiesta a carico dell'hardware, all'allocazione intenzionale durante il normale caricamento delle variabili mediante le direttive del compilatore. In ogni caso, si pone il problema di garantire la coerenza.

Una seconda tecnica per celare la latenza è il **prefetching**. Se è possibile caricare un dato prima di doverlo utilizzare, la sua lettura può essere sovrapposta alla normale esecuzione e così il dato sarà già disponibile al momento della richiesta. Il prefetching può essere automatico o sotto il controllo del programma. Il caricamento nella cache di un'intera linea, e non solo della parola referenziata, punta sul fatto che probabilmente anche le parole successive verranno richieste molto presto.

Vediamo com'è possibile controllare il prefetching in modo esplicito: quando il compilatore si rende conto che avrà bisogno di certi dati, può inserire un'istruzione esplicita per il loro caricamento e anticiparla in modo tale che i dati siano già disponibili quando necessario. Ciò comporta da parte del compilatore una conoscenza totale della macchina sottostante e della sua temporizzazione, come anche il pieno controllo della localizzazione dei dati. Una siffatta istruzione speculativa di LOAD funziona bene solo se si è certi che i dati saranno richiesti; provocare un errore di pagina a seguito di una LOAD che risulta poi inutile si rivela molto penalizzante.

Una terza tecnica utile per nascondere la latenza è il multithreading, come abbiamo già visto. Se la commutazione tra processi può essere resa abbastanza veloce, per esempio dotando ciascun processo di una propria mappa di memoria e di propri registri, allora l'hardware può commutare velocemente da un thread bloccato in attesa di dati distanti verso un thread in grado di proseguire l'esecuzione. Nel caso peggiore la CPU esegue un'istruzione del primo thread, una del secondo, e così via. Così facendo, si è sicuri di mantenere la CPU occupata anche in presenza di latenze di memoria molto lunghe cui andrebbero soggetti i singoli thread.

Una quarta tecnica che aiuta a nascondere la latenza è quella delle scritture non bloccanti. In genere l'esecuzione di un'istruzione STORE causa il blocco della CPU fino al completamento dell'operazione. L'avvio di una scrittura non bloccante consente invece la prosecuzione del programma anche se l'operazione di memoria non è stata ancora completata. La prosecuzione dell'esecuzione durante un'istruzione di LOAD è più difficile da realizzare, ma è possibile in caso di esecuzione fuori sequenza.

8.5 Grid computing

Molte delle sfide attuali nelle scienze, nell'ingegneria, nell'industria, nell'ambiente e in altri settori, sono di natura interdisciplinare e di vaste dimensioni. La loro risoluzione richiede esperienza, abilità, conoscenza, disponibilità di mezzi, software e dati provenienti da numerose organizzazioni, spesso ubicate in paesi diversi. Vediamo alcuni esempi.

1. Il progetto di una missione su Marte.
2. Un consorzio di aziende che costruisce un prodotto molto complesso (per esempio, una diga o un aeroplano).
3. Una squadra di soccorso internazionale che coordina gli aiuti dopo una catastrofe naturale.

Alcune di queste cooperazioni sono a lungo termine, altre a breve termine, ma tutte presentano come caratteristica comune il coinvolgimento di organizzazioni separate cui viene richiesto di partecipare con le proprie risorse e procedure di lavoro per raggiungere insieme lo scopo condiviso.

Fino a qualche tempo fa, la condivisione di dati e risorse tra organizzazioni che utilizzavano sistemi operativi, basi di dati e protocolli differenti era molto difficile. Ciononostante, il bisogno crescente di cooperazione su larga scala tra organizzazioni internazionali ha portato allo sviluppo di sistemi per la connessione di calcolatori molto distanti e altrimenti separati, che prendono il nome di tecnologia **grid**, o **grid computing**. In un certo senso tale tecnologia è lo stadio successivo della Figura 8.1 e può essere pensata come un cluster internazionale molto grande, lascamente connesso ed eterogeneo.

Lo scopo della tecnologia grid è fornire un'infrastruttura tecnica che permetta la condivisione di un obiettivo comune tra un gruppo di organizzazioni che formano così un'**organizzazione virtuale**. Tale organizzazione deve essere flessibile, deve poter comprendere un numero di partecipanti molto grande e gestire le loro variazioni, deve consentire ai propri membri di lavorare in accordo quando lo ritengono opportuno, mentre deve garantire loro di conservare il controllo delle proprie risorse con il grado di esclusività desiderato. I ricercatori del settore grid stanno lavorando allo sviluppo di servizi, strumenti e protocolli per rendere queste organizzazioni virtuali in grado di funzionare.

La tecnologia grid è intrinsecamente multilaterale, perché ha molti partecipanti di pari grado e può essere utile metterla in contrapposizione con gli altri modelli di calcolo esistenti. Nel modello client-server una transazione coinvolge due agenti: il server, che offre un certo servizio, e il client, che desidera avvalersene. Un esempio tipico di modello client-server è il Web, in cui gli utenti si rivolgono ai server web per trovare le informazioni. La tecnologia grid differisce anche dalle applicazioni *peer-to-peer* ("da pari a pari"), le quali consentono lo scambio diretto di file tra coppie di utenti (l'e-mail ne è un esempio comune). Poiché la tecnologia grid differisce da tutti questi modelli, richiede nuovi protocolli e tecnologie.

La tecnologia grid ha bisogno di poter accedere a un gran numero di risorse. Ogni risorsa risiede su di un sistema specifico e appartiene a un'organizzazione che decide con quali modalità renderla disponibile, quando e a chi. In astratto si può dire che la tecnologia grid si occupa dell'accesso alle risorse e della loro gestione.

La Figura 8.50 mostra un modo possibile di modellare una tecnologia grid come una gerarchia di livelli. Il **livello struttura** (*fabric layer*) alla base della gerarchia comprende l'insieme dei componenti costitutivi dell'infrastruttura, ovvero CPU, dischi, reti e sensori (per quanto riguarda l'hardware), e i dati e i programmi (per il lato software). Queste sono le risorse messe a disposizione in maniera controllata dalla tecnologia grid.

Il livello **risorse** (*resource layer*) attiene alla gestione delle risorse individuali. In molti casi, una risorsa che partecipa a un sistema grid viene gestita da un processo locale che consente agli utenti distanti di accedervi in modo controllato. Questo livello fornisce ai livelli superiori un'interfaccia uniforme per ispezionare le caratteristiche e lo stato delle risorse individuali, per monitorarle e per usarle in modo sicuro.

Il livello successivo è quello **di raccolta** (*collective layer*) che gestisce gruppi di risorse. Una delle sue funzioni è la scoperta di risorse, tramite cui un utente può localizzare cicli di CPU disponibili, spazio su disco o dati specifici. Il livello di raccolta può servirsi di directory o di altri database per gestire queste informazioni, e può anche fornire un servizio di mediazione per trovare le corrispondenze migliori tra fornitori di servizi e loro utenti, eventualmente allocando le risorse poco reperibili tra gli utenti in competizione per accaparrarsele. Il livello di raccolta è responsabile anche della duplicazione dei dati, dell'ammissione di nuovi membri e risorse e del mantenimento del database delle politiche che specifica quali utenti possono usare quali risorse.

Al di sopra dei precedenti si trova il **livello applicativo** (*application layer*), che si avvale dei livelli inferiori per procurarsi le credenziali che provano il suo diritto all'uso di certe risorse, a richiederne l'impiego, a monitorarne i progressi, a trattarne gli errori e a notificare i loro risultati agli utenti.

La caratteristica fondamentale di una buona tecnologia grid è la sicurezza. I proprietari delle risorse sono molto insistenti nel richiedere un controllo ravvicinato delle loro risorse e vogliono sapere chi le usa, per quanto tempo e con quale intensità. Nessuna organizzazione renderebbe disponibili le proprie risorse con una tecnologia grid senza un buon livello di sicurezza. D'altro canto, se un utente dovesse avere un account protetto da password su ogni singolo computer che intende usare, l'uso della tecnologia grid diverrebbe insopportabilmente scomodo. Dunque è necessario sviluppare un modello di sicurezza che risponda a queste richieste adeguatamente.

Figura 8.50 Livelli della tecnologia grid.

Una delle caratteristiche fondamentali del modello di sicurezza è richiedere una sola autenticazione per l'ingresso nel sistema (*single sign on*). Il primo passo per l'uso di un tecnologia grid è venire autenticati e acquisire delle credenziali, cioè un documento con firma digitale che individua l'utente per conto del quale va svolto il lavoro. Le credenziali possono essere usate come delega, così, se un'elaborazione ha bisogno di creare sotto-elaborazioni, anche i processi figli possono essere identificati. Quando si presenta una credenziale a una macchina distante, questa deve verificarla con il proprio meccanismo di sicurezza locale. Per esempio, nei sistemi UNIX gli utenti sono identificati tramite un identificatore di 16 bit, ma altri sistemi usano schemi diversi. Infine la tecnologia grid deve prevedere alcuni meccanismi per la specifica, il mantenimento e l'aggiornamento delle politiche di accesso.

L'interoperatività tra organizzazioni e macchine diverse richiede l'adozione di standard, sia in termini di servizi offerti, sia di protocolli usati per il loro accesso. La comunità grid ha creato un'organizzazione, il Global Grid Forum, proprio per gestire il processo di standardizzazione. Un esito del forum è stato la produzione di uno schema chiamato **OGSA** (*Open Grid Services Architecture*) per la sistemazione dei diversi standard in via di sviluppo. Laddove possibile, si utilizzano standard già esistenti: per esempio è stato usato WSDL (*Web Services Definiton Language*) per la descrizione dei servizi OGSA. I servizi attualmente in corso di standardizzazione appartengono alle seguenti otto vaste categorie, ma ne verranno certamente create di nuove.

1. Servizi di infrastruttura (per la comunicazione tra risorse).
2. Servizi di gestione delle risorse (prenotazione e messa a disposizione delle risorse).
3. Servizi per i dati (trasferimento e duplicazione dei dati laddove richiesto).
4. Servizi di contesto (descrizione delle risorse richieste e delle politiche di utilizzo).
5. Servizi informativi (ottenimento delle informazioni circa la disponibilità di risorse).
6. Servizi di amministrazione interna (garanzia della qualità di servizio specificata).
7. Servizi di sicurezza (rispetto delle politiche di sicurezza).
8. Esecuzione di servizi amministrativi (gestione dello scheduling dei compiti).

Resterebbe ancora molto da dire su questo argomento, ma per ragioni di spazio non possiamo dilungarci ulteriormente. Per una trattazione approfondita delle tecnologie grid rimandiamo ai riferimenti (Berman et al., 2003; Foster e Kesselman, 2003; Foster et al., 2002).

8.6 Riepilogo

È sempre più difficile rendere più veloci i computer limitandosi a incrementare la loro frequenza di clock, a causa dei problemi della dissipazione del calore e di altri fattori. I progettisti cercano invece di sfruttare il parallelismo per incrementare le prestazioni. Il parallelismo può essere introdotto a molti livelli, a partire dai più bassi, dove gli elementi dell'elaborazione sono legati molto strettamente, fino a quelli più alti, dove sono legati molto debolmente.

A livello più basso c'è il parallelismo nel chip. In primo luogo esiste il parallelismo a livello delle istruzioni, in cui un'istruzione o una sequenza d'istruzioni emettono diverse operazioni che possono essere svolte in parallelo da unità funzionali distinte. Una seconda forma di parallelismo nel chip è il multithreading, in cui la CPU può commutare continuamente tra un thread e l'altro in base al tipo d'istruzione, creando così un multiprocessore virtuale. Una terza forma di parallelismo nel chip è il multiprocessore a chip singolo, in cui due o più core disposti all'interno dello stesso chip sono in grado di girare in parallelo.

A livello superiore troviamo i coprocessori, che risiedono in genere su schede a innesto e mettono a disposizione potenza di calcolo ausiliaria per certi calcoli specializzati, come l'elaborazione di protocolli di rete o l'elaborazione multimediale. Questi processori aggiuntivi sollevano la CPU da una porzione del lavoro e le permettono così di svolgere altri calcoli mentre si occupano dei loro compiti specializzati.

A livello successivo troviamo i multiprocessori con memoria condivisa. Questi sistemi contengono due o più CPU vere e proprie che condividono una memoria. I multiprocessori UMA comunicano attraverso un bus condiviso (che effettua snooping), un commutatore crossbar o una rete a commutazione multilivello. Sono caratterizzati dal fatto che garantiscono un tempo di accesso uniforme alle diverse locazioni di memoria. Anche i multiprocessori NUMA mostrano a tutti i processi uno spazio degli indirizzi condiviso, ma in questo caso gli accessi a distanza impiegano un tempo considerevolmente maggiore rispetto a quelli locali. Infine, i multiprocessori COMA costituiscono un'ulteriore variante di multiprocessori in cui le linee di cache vengono trasferite all'interno del sistema su richiesta, ma, diversamente da altri tipi di progetto, non appartengono ad alcuna locazione in particolare.

I multicomputer sono sistemi dotati di molte CPU che non condividono memoria. Ciascuna CPU ha la sua memoria privata e comunica con le altre tramite scambio di messaggi. Gli MPP, tra cui ricordiamo BlueGene/L di IBM, sono multicomputer molto grandi dotati di reti di comunicazione specializzate. I cluster sono sistemi più semplici costruiti a partire da componenti comunemente disponibili sul mercato, come le macchine che fanno funzionare Google.

I multicomputer sono spesso programmati avvalendosi di pacchetti software per lo scambio di messaggi quali MPI. Un approccio alternativo consiste nell'usare la condivisione di memoria a livello applicativo; ne sono alcuni esempi un sistema DSM basato su pagine, lo spazio delle tuple di Linda o gli oggetti di Orca o di Globe. DSM simula la memoria condivisa a livello della pagine, rendendo il sistema simile a una macchina NUMA, pur penalizzando molto di più gli accessi a distanza.

Infine, a livello più alto e legato più debolmente, c'è il grid computing. Si tratta di sistemi che raggruppano intere organizzazioni di utenti Internet, disposti a condividere potenza di calcolo, dati e altre risorse.

Problemi

1. Le istruzioni del Pentium possono raggiungere i 17 byte di lunghezza. Il Pentium è una CPU VLIW?
2. Quali sono i valori saturati di 96, –9, 300 e 256 se l'intervallo di saturazione è 0–255?
3. Si stabilisca se le seguenti istruzioni di TriMedia sono lecite e, in caso negativo, si indichi perché non lo sono.
   (a) Somma intera, sottrazione intera, caricamento, somma in virgola mobile, caricamento immediato.
   (b) Sottrazione intera, moltiplicazione intera, caricamento immediato, scorrimento, scorrimento.
   (c) Caricamento immediato, somma in virgola mobile, moltiplicazione in virgola mobile, salto, caricamento immediato.
4. Le Figure 8.7(d) e 8.7(e) mostrano 12 cicli d'istruzioni. Per ciascuna delle due sequenze, si indichi che cosa succede nei tre cicli successivi.
5. In una certa CPU, un'istruzione che causa un fallimento nella cache di primo livello, ma che ha successo nella cache di secondo livello, richiede in tutto k cicli. Se si usa il multithreading per nascondere i fallimenti di cache di primo livello, quanti thread bisogna eseguire contemporaneamente per evitare cicli di inattività con il multithreading a grana fine?
6. Una mattina l'ape regina di un certo alveare convoca le api operaie e affida loro, come compito per la giornata, la raccolta di nettare di calendula. Quindi le operaie volano in direzioni diverse in cerca di calendule. Si tratta di un sistema SIMD o MIMD?
7. Quando abbiamo esaminato i modelli di consistenza della memoria, abbiamo detto che un modello di consistenza è una specie di contratto tra il software e la memoria. Perché è necessario un tale contratto?
8. Si consideri un multiprocessore con bus condiviso. Che cosa succede se due processori cercano di accedere alla memoria globale esattamente nello stesso istante?
9. Si supponga che, per qualche ragione tecnica, una cache possa fare lo snooping solo delle linee degli indirizzi, e non delle linee dati. Questa limitazione avrebbe ripercussioni sul protocollo write through?
10. Si prenda in considerazione un semplice modello di sistema multiprocessore senza caching basato sul bus, in cui ogni quattro istruzioni una accede alla memoria e l'accesso alla memoria occupa il bus per un tempo pari all'esecuzione di un'istruzione. Quando il bus è occupato, le CPU richiedenti vengono poste in una coda d'attesa FIFO. Quanto un sistema di 64 CPU risulterà più veloce rispetto a un sistema con una sola CPU?
11. Il protocollo MESI di coerenza delle cache ha quattro stati, mentre altri protocolli write-back di coerenza delle cache ne hanno solo tre. Quale dei quattro stati di MESI può essere sacrificato e con quali conseguenze? Se si dovessero scegliere solo tre stati, quali sarebbero?
12. Il protocollo MESI di coerenza delle cache dà luogo a situazioni in cui una linea di cache richiede una transazione sul bus pur essendo presente nella cache locale? Se sì, si spieghi perché.

13. Ci sono n CPU collegate a un bus comune. La probabilità che una CPU cerchi di usare il bus in un determinato ciclo è p. Qual è la probabilità che durante un ciclo
    a. il bus resti inoperoso (0 richieste)
    b. venga fatta una sola richiesta
    c. vengano fatte più richieste?
14. Quanti commutatori crossbar ci sono in un E25K di Sun Fire completo?
15. Se si interrompe il collegamento tra i commutatori 2A e 3B della rete omega, quali elementi ne risultano scollegati?
16. Gli *hot spot* ("punti caldi") sono le locazioni di memoria cui si accede frequentemente e costituiscono un problema grave nelle reti a commutazione multilivello. Sono un problema anche nei sistemi basati su bus?
17. Una rete di commutazione omega collega 4096 CPU RISC, ciascuna con ciclo di 60 ns, a 4096 moduli di memoria infinitamente veloci. Gli elementi che effettuano la commutazione introducono un ritardo di 5 ns. Quanti intervalli di ritardo sono necessari a un'istruzione LOAD?
18. Si consideri una macchina che usa una rete di commutazione omega come quella mostrata nella Figura 8.29. Se il modulo di memoria i contiene il programma e lo stack del processore i, si proponga una piccola modifica della topologia che produca un grosso cambiamento delle prestazioni (RP3 di IBM e Butterfly di BBN usano questa topologia modificata). Quale svantaggio comporta la nuova topologia rispetto a quella originale?
19. In un multiprocessore NUMA, un riferimento alla memoria locale impiega 20 ns, mentre un riferimento a distanza impiega 120 ns. Un certo programma effettua N riferimenti a memoria durante la sua esecuzione, l'1% dei quali avviene verso la pagina P. La pagina P si trova inizialmente a distanza e ci vogliono C secondi per copiarla localmente. In quali condizioni conviene copiare la pagina localmente, in assenza di un suo utilizzo significativo da parte di altri processori?
20. Si consideri un multiprocessore CC-NUMA simile a quello della Figura 8.31, ma con 512 nodi dotati di 8 MB ciascuno. Se le linee di cache sono di 64 byte, qual è la percentuale d'informazioni accessorie necessarie alle directory? L'incremento del numero di nodi aumenta, diminuisce o lascia invariata questa percentuale?
21. Si calcoli il diametro di rete delle topologie mostrate nella Figura 8.35.
22. Si calcoli il grado di tolleranza agli errori delle topologie mostrate nella Figura 8.35. Il grado di tolleranza è definito come il massimo numero di collegamenti che è possibile interrompere senza separare la rete in due componenti sconnessi.
23. Si consideri la topologia a doppio toro della Figura 8.35(f) espansa alle dimensioni $k \times k$. Qual è il diametro della rete risultante? Suggerimento: distinguere i casi con k pari e k dispari.
24. Una rete d'interconnessione ha la forma di un cubo $8 \times 8 \times 8$. Ogni collegamento ha una larghezza di banda full duplex di 1 GB/s. Qual è la larghezza di banda di bisezione della rete?
25. La legge di Amdhal limita l'incremento potenziale raggiungibile da un calcolatore parallelo. Si calcoli, in funzione di f, l'incremento massimo raggiungibile per un numero di CPU che tende a infinito. Quali conseguenze ha questo limite per $f = 0{,}1$?

26. La Figura 8.49 mostra la buona scalabilità di una griglia in contrapposizione alla cattiva scalabilità di un bus. Se ogni bus o collegamento ha una larghezza di banda *b*, si calcoli la larghezza di banda media per CPU in ciascuno dei quattro casi della figura. Si scalino quindi tutti i sistemi alla dimensione di 64 CPU e si ripeta il calcolo. Qual è il limite se il numero di CPU tende a infinito?

27. Abbiamo trattato nel capitolo tre varianti di `send`: sincrona, bloccante e non bloccante. Si fornisca un quarto metodo che sia simile alla variante bloccante, ma che abbia proprietà leggermente diverse. Si elenchino vantaggi e svantaggi della versione proposta valutati rispetto alla `send` bloccante.

28. Si consideri un multicomputer che usa una rete che fa broadcast hardware come Ethernet. Perché è importante il rapporto tra il numero di operazioni di lettura (che non modificano lo stato interno delle variabili) e il numero di operazioni di scrittura (che modificano lo stato interno delle variabili)?

Capitolo 9

Bibliografia e letture consigliate

Negli otto capitoli precedenti abbiamo affrontato un gran numero di argomenti in modo più o meno dettagliato. Questo capitolo è concepito come supporto per quei lettori che vogliano approfondire lo studio dell'organizzazione dei calcolatori. Il Paragrafo 9.1 contiene una lista di letture consigliate e il successivo raccoglie in ordine alfabetico tutti i riferimenti bibliografici citati nel testo.

9.1 Letture di approfondimento

Indichiamo alcune letture di approfondimento elencate per argomento. Si tratta per lo più di libri di testo, tutorial o articoli.

9.1.1 Lavori introduttivi

Borkar, "Getting Gigascale Chips"

La legge di Moore probabilmente sarà ancora valida per almeno un decennio e i chip cresceranno fino a contenere miliardi di transistor. I chip così grandi sono al contempo una risorsa e una sfida. Questo articolo, redatto da uno dei principali ricercatori di Intel, illustra le sfide che bisognerà affrontare in futuro e che sono legate per esempio alla dissipazione del calore, alla crescita di resistenza e capacità di conduttori sempre più miniaturizzati, ecc. L'autore sostiene che le soluzioni future di baseranno sul multithreading, sui multiprocessori nel chip e su strutture di memoria migliorate, piuttosto che sul mero incremento della frequenza di clock.

Colwell, *The Pentium Chronicles*

Robert Colwell era a capo del gruppo di lavoro che progettò il Pentium. Nel suo libro parla dei retroscena della progettazione di questo chip, a partire dalle persone coinvolte e dalla loro passione, per finire alla politica.

Hamacher et al., *Computer Organization, V edizione*

Un libro di testo tradizionale sull'organizzazione dei calcolatori che descrive CPU, memorie, I/O, aritmetica e periferiche. Gli esempi trattati sono il 68000 e il PowerPC.

Heath, *Embedded Systems Design*

Oggigiorno, quasi ogni dispositivo elettrico che costi più di 50 euro contiene un calcolatore. Questo libro ha come soggetto i sistemi integrati; la trattazione comincia dagli argomenti basilari, come processori integrati, memorie e periferiche, per poi affrontare interfacce, sistemi operativi in tempo reale, software e debugging.

Hennessy e Patterson, *Computer Architecture. A Quantitative Approach, III edizione*

Questo voluminoso libro di testo si spinge nei dettagli più minuziosi della progettazione di processori e memorie. Particolare attenzione è rivolta al raggiungimento di prestazioni elevate mediante il parallelismo e l'uso di pipeline. Il lettore che desideri sapere tutto sulla progettazione di una CPU di fascia alta non troverà lettura più adeguata.

Null e Lobur, *The Essentials of Computer Organization and Architecture*

Un altro testo sull'organizzazione dei calcolatori che copre gli stessi argomenti di questo libro, ma con minor dettaglio.

Patterson e Hennessy, *Computer Organization and Design, III edizione*

La terza edizione non comprende più le oltre mille pagine della precedente edizione. Infatti una buona porzione del testo è stata trasferita sul CD-ROM allegato. Il materiale rimasto nel libro tratta vari aspetti dell'architettura dei calcolatori, tra cui l'aritmetica, le prestazioni, il percorso dati, le pipeline, le memorie, le periferiche e i cluster. Benché il Pentium 4 sia descritto in diversi punti del testo, l'esempio principale della trattazione è il processore MIPS, progettato da Hennessy e inserito nel 1985 nella prima macchina RISC commerciale mai costruita.

Price, "A History of Calculating Machines"

Anche se il primo computer moderno si deve a Babbage nel XIX secolo, il genere umano ha sempre usato il calcolo sin dagli albori della civiltà. Questo articolo affascinante ripercorre la storia della matematica, dei calendari e del calcolo dal 3000 a.C. fino all'inizio del XX secolo.

Slater, *Portraits in Silicon*

Perché Dennis Ritchie rinunciò a completare la sua tesi di Ph.D. ad Harvard? Perché Steve Job è diventato vegetariano? Le risposte si trovano in questo libro affascinante, contenente 34 brevi biografie di persone che hanno fatto la storia dell'industria dei computer, da Charles Babbage fino a Donald Knuth.

Stallings, *Architettura e organizzazione dei calcolatori. Progetto e prestazioni. VI edizione*

Un testo generale sull'architettura dei calcolatori. Alcuni degli argomenti trattati in questo libro sono contenuti anche nel libro di Stallings.

Wilkes, "Computers Then and Now"

Una storia dei calcolatori dal 1946 al 1968 scritta dal punto di vista personale di Maurice Wilkes, pioniere della progettazione dei calcolatori e inventore della microprogrammazione. Racconta le dispute che ci furono agli inizi tra i "cadetti dello spazio", fautori della programmazione automatica (con i compilatori pre-FORTRAN), e i tradizionalisti, che preferivano programmare in ottale.

9.1.2 Organizzazione dei calcolatori

Buchanan e Wilson, *Advanced PC Architecture*

Questo libro tratta molti argomenti legati ai processori, ai bus (PCI, SCSI e USB), alle porte (parallele, seriali e per videogiochi) e ad altri componenti dei PC, benché lo faccia in modo un po' disordinato.

Ng, "Advances in Disk Technology: Performance Issues"

Sono almeno vent'anni che gli esperti predicono la scomparsa dei dischi magnetici, ma finora sono rimasti in circolazione. Questo articolo sostiene che i loro rapidi progressi tecnologici li faranno sopravvivere anche nei prossimi anni a venire.

Messmer, *The Indispensable PC Hardware Book, IV edizione*

Le 1296 pagine (divise in 37 capitoli e 7 appendici) di questo libro potrebbero essere più o meno indispensabili, ma sicuramente lo rendono davvero enorme. Vi si può trovare con dettaglio infinitesimo praticamente tutto ciò che ha a che fare con processori 80x86, memorie, bus, chip di supporto e periferiche. È una lettura indicata a chi abbia già letto e digerito il libro (seguente) di Norton e Goodman e voglia spostarsi a livello di dettaglio tecnico successivo.

Norton e Goodman, *Inside the PC, VIII edizione*

Molti dei libri che trattano l'hardware dei PC sono rivolti agli ingegneri elettronici e risultano di difficile lettura per chi è dotato di una cultura più orientata al software. Questo libro è differente, perché spiega l'hardware in un modo che coniuga il rigore tecnico e l'accessibilità dei contenuti. Gli argomenti comprendono CPU, memoria, bus, dischi, dispositivi di I/O, PC portatili, rete, ecc. Un testo raro dalle qualità apprezzabili.

Robinson, "Toward the Age of Smarter Storage"

La memoria dei calcolatori ha fatto passi da gigante dai tempi delle memorie a nuclei di ferrite e delle schede perforate. Questo breve articolo è dedicato alla tecnologia delle memorie di una volta, al suo presente e alle tendenze per il futuro.

Scheible, "A Survey of Storage Options"

Un'altra rassegna sulla memorizzazione, questa volta incentrata sulla realtà attuale. Sono presentati i vari tipi di RAM, memorie flash, a nastro, hard disk, floppy disk, CD e DVD.

Stan e Skadron, "Power-Aware Computing"

I calcolatori stanno diventando letteralmente troppo potenti, nel senso che assorbono troppa energia e ciò costituisce un problema nel settore dei dispositivi portatili. Questo articolo è il contributo introduttivo dei curatori di un numero speciale della rivista *IEEE Computer Magazine*, dedicato al calcolo a basso assorbimento energetico.

Triebel, *The 80386, 80486, and Pentium Processor*

Questo libro è difficile da classificare, perché tratta dell'hardware, del software e delle interfacce. Lo annoveriamo tra i libri sull'hardware perché il suo autore è un progettista di Intel. Il testo tratta di processori, memorie, dispositivi di I/O e interfacce dei chip 80x86, ma anche della loro programmazione in linguaggio assemblativo. Benché sia costituito solo da 915 pagine, contiene almeno lo stesso quantitativo di materiale del libro di Messmer in virtù di un formato di pagina maggiore.

9.1.3 Livello logico digitale

Floyd, *Digital Fundamentals, VIII edizione*

Per i lettori con background hardware che vogliano approfondire lo studio del livello logico digitale, questo libro enorme, riccamente illustrato in quadricromia, costituisce una perla preziosa. I diversi capitoli coprono logica combinatoria, dispositivi logici programmabili, flip-flop, registri di scorrimento, memorie, interfacce e molto altro.

Mano e Kime, *Logic and Computer Design Fundamentals, III edizione*

Questo libro non ha certo la struttura elegante del libro di Floyd, ma è comunque un buon riferimento per il livello logico digitale. Tratta circuiti combinatori e sequenziali, registri, memorie, progettazione di CPU e dell'I/O.

Mayhew e Krishnan, "PCI Express and Advanced Switching"

PCI Express quasi sicuramente rimpiazzerà il bus PCI nel prossimo futuro e questo articolo è un breve tutorial che ne descrive livelli, controllo di flusso, canali virtuali, commutazione e instradamento.

Mazidi e Mazidi, *The 80x86 IBM PC and Compatible Computers, IV edizione*

I lettori interessati alla comprensione di tutti i chip contenuti in un PC troveranno in questo libro interi capitoli sui chip più importanti, insieme ad altre informazioni sull'hardware dei PC IBM e alla programmazione in linguaggio assemblativo.

Roth, *Fundamentals of Logic Design*

Questo testo copre gli aspetti basilari della progettazione logica, dall'algebra di Boole alle porte logiche, ai contatori, ai sommatori, ai flip-flop e ad altri tipi di circuiti combinatori e sequenziali.

9.1.4 Livello di microarchitettura

Burger e Goodman, "Billion-Transistor Architectures: There and Back Again"

Se nel 1997 vi avessero chiesto di progettare un chip con un miliardo di transistor, che tipo di microarchitettura avreste ideato? È la stessa domanda che venne posta nel settembre del 1997 a sette affermati ricercatori di architetture, le cui risposte vennero pubblicate nella rivista *IEEE Computer Magazine*. Questo articolo confronta le loro predizioni con lo stato attuale della ricerca.

Handy, *The Cache Memory Book, II edizione*

La progettazione delle cache è così importante che oggi esistono libri interamente dedicati all'argomento. Questo testo mette a confronto le cache logiche con quelle fisiche, le politiche write-through con quelle write-back, le cache unificate con quelle specializzate, tratta la dimensione di linea e altre questioni software. Contiene inoltre un capitolo sui multiprocessori con cache coerente.

Johnson, *Superscalar Microprocessor Design*

I lettori interessati ai dettagli della progettazione di CPU superscalari possono cominciare da questo libro. Tratta il fetch e la decodifica delle istruzioni, l'emissione fuori sequenza, la rinomina di registri, le stazioni di prenotazione, la predizione dei salti e altro ancora.

Shriver e Smith, "The Anatomy of a High-Performance Microprocessor"

Uno studio valido e dettagliato dei moderni chip di CPU a livello di microarchitettura, che esamina il chip K6 di AMD, un clone del Pentium, con particolare attenzione a pipeline, scheduling delle istruzioni e ottimizzazione delle prestazioni.

Sima, "Superscalar Instruction Issue"

Le istruzioni superscalari assumono sempre maggiore importanza nelle CPU. Nel testo ci si è occupati di alcuni aspetti del problema, come la rinomina e l'esecuzione speculativa, e questo articolo esamina molte altre questioni attinenti l'argomento.

Wilson, "Challenges and Trends in Processor Design"

La progettazione dei processori è una disciplina morta? Certo che no. Sei architetti di CPU di chiara fama, che lavorano per Sun, Cyrix, Motorola, Mips, Intel, e Digital, riferiscono il loro pensiero sulle prospettive della progettazione di CPU nei prossimi anni. Oggi è una lettura istruttiva, nel 2008 sarà una lettura divertente.

9.1.5 Livello di architettura dell'insieme d'istruzioni

Antonakos, *The Pentium Microprocessor*

I primi nove capitoli di questo libro trattano la programmazione del Pentium in linguaggio assemblativo, gli ultimi due l'hardware. Il testo è corredato di numerosi frammenti di codice e analizza il BIOS estesamente.

Ayala, *The 8051 Microcontroller, III edizione*

Un buon punto di partenza per i lettori che vogliono imparare a programmare l'8051.

Bryant e O'Hallaron, *Computer Systems. A Programmer's Perspective*

Un libro che copre (anche se in modo un po' disorganizzato) molti temi relativi a livello ISA, tra cui l'aritmetica, i diversi tipi d'istruzioni, il controllo del flusso e l'ottimizzazione dei programmi.

Paul, *SPARC Architecture, Assembly Language, Programming, and C*

Meraviglia delle meraviglie, ecco un libro sulla programmazione in linguaggio assemblativo che non parla della linea Intel 80x86. Tratta invece dello SPARC e della sua programmazione.

Weaver e Germond, *The SPARC Architecture Manual*

La diffusione su scala mondiale dell'industria dei computer attribuisce sempre più importanza agli standard, perciò è fondamentale imparare a conoscerli. Questo è relativo alla Versione 9 dello SPARC ed è abbastanza rappresentativo degli altri standard; inoltre fornisce molte informazioni sugli SPARC a 64 bit.

9.1.6 Livello macchina del sistema operativo

Hart, "Win32 System Programming"

A differenza di quasi tutti gli altri libri su Windows, questo non si sofferma sull'interfaccia grafica. Analizza invece le chiamate di sistema e il loro utilizzo per l'accesso ai file, la gestione di memoria e dei processi, la comunicazione tra processi, l'uso dei thread, dell'I/O, ecc.

Jacob e Mudge, "Virtual Memory: Issues of Implementation"

Una buona introduzione alla memoria virtuale. Spiega le varie tabelle delle pagine, le strutture TLB e illustra queste idee usando i processori MIPS, PowerPC e Pentium.

McKusick et al., *Design and Implementation of the 4.4 BSD Operating System*

A differenza di molti libri su UNIX, questo inizia con una foto dei quattro autori partecipanti alla conferenza USENIX. Tre di loro hanno scritto gran parte di BSD 4.4 e sono quindi altamente qualificati per spiegarne il funzionamento interno. Il libro analizza le chiamate di sistema, i processi, l'I/O e contiene una sezione sulle reti particolarmente ben riuscita.

Ritchie e Thompson, "The UNIX Time-Sharing System"

L'articolo originale su UNIX costituisce una lettura molto utile. Da questo piccolo seme è nato un grande sistema operativo.

Russinovich e Solomon, *Inside Microsoft Windows, IV edizione*

La scelta migliore per chi voglia comprendere il funzionamento interno di Windows. Tratta l'architettura e i meccanismi di sistema, i processi, i thread, la gestione di memoria, la sicurezza, l'I/O, la cache, il file system e altri argomenti. Il libro è concepito come studio approfondito rivolto agli studenti di informatica e ai professionisti del mondo dell'informatica.

Tanenbaum e Woodhull, *Operating Systems: Design and Implementation, II edizione*

Molti libri sui sistemi operativi si limitano a trattare la teoria, mentre questo affianca alla necessaria trattazione teorica l'analisi del codice di MINIX, un vero e proprio sistema operativo di tipo UNIX che gira sui PC IBM e su altri computer. Il codice sorgente è ampiamente commentato in appendice.

9.1.7 Livello del linguaggio assemblativo

Levine, *Linkers and Loaders*

Un libro per gli appassionati di linker e loader, dei vari formati di file oggetto, della differenza tra collegamento statico e dinamico e dei diversi formati delle librerie.

Saloman, *Assemblers and Loaders*

Tutto ciò che bisogna sapere sul funzionamento degli assemblatori a una o due passate, sui linker e sui loader. Tratta anche le macro e l'assemblaggio condizionale.

9.1.8 Architetture per il calcolo parallelo

Adve e Gharachorloo, "Shared Memory Consistency Models: A Tutorial"

Molti computer moderni, e in special modo i multiprocessori, si fondano su un modello di memoria più debole di quello a consistenza sequenziale. Questo lavoro espone vari modelli e spiega il loro funzionamento. Vengono anche avvalorati o confutati diversi miti riguardanti la memoria debolmente consistente.

Comer, *Network Systems Design*

La prima parte del libro riguarda l'elaborazione tradizionale dei pacchetti di rete, mentre la seconda parte introduce i processori di rete e descrive il loro scopo, le loro architetture e i compromessi della loro progettazione. La terza parte descrive il processore di rete Agere scelto come esempio tipo della trattazione.

Dally e Towles, *Principles and Practices of Interconnection Networks*

Il testo di riferimento sulle reti d'interconnessione che affronta le reti a farfalla, le reti toroidali e le reti non bloccanti. Altri capitoli sono dedicati a instradamento, controllo del flusso, bufferizzazione, deadlock e altri argomenti correlati.

Dongarra et al., *The Sourcebook of Parallel Computing*

La programmazione dei multiprocessori e dei cluster è profondamente diversa dalla programmazione sequenziale. Il libro riporta il contributo di sette esperti della programmazione parallela alla trattazione di numerosi aspetti del calcolo parallelo, tra cui architetture, tecnologie software, algoritmi e applicazioni.

Hill, "Multiprocessors Should Support Simple Memory-Consistency Models"

L'indebolimento della semantica della memoria è un argomento molto discusso e controverso nella progettazione delle memorie dei multiprocessori. Modelli più deboli permettono alcune ottimizzazioni hardware, come i riferimenti a memoria fuori sequenza, ma rende la programmazione più difficile. Gli autori di questo articolo discutono diversi aspetti rilevanti della coerenza e concludono che una memoria indebolita causa più problemi di quanti ne risolva.

Hwang e Xu, *Scalable Parallel Computing*

Un resoconto completo e accessibile sul calcolo parallelo (sia del software, sia dell'hardware). Tra gli argomenti ci sono i multiprocessori UMA e NUMA, i sistemi MPP e COW, quelli a scambio di messaggi e la programmazione con parallelismo dei dati.

Lawton, "Will Network Processor Units Live up to Their Promise?"

I processori di rete promettono l'incremento della velocità di trattamento dei pacchetti, ma il loro successo non è scontato. In questo articolo, l'autore rivolge la sua attenzione alla tecnologia e ad altri fattori che possono influenzare, in positivo o in negativo, il mantenimento di questa promessa.

McKnight et al., "Wireless Grids"

La tecnologia grid ha appena visto la luce e già si affaccia all'orizzonte la prossima generazione: la tecnologia grid senza fili. Al pari dei sistemi grid convenzionali, quelli senza fili intendono favorire la condivisione di risorse tra organizzazioni per la creazione di organizzazioni virtuali, ma si avvalgono di tecnologie senza fili per condividere le risorse tra utenti muniti di dispositivi portatili.

Pfister, *In Search of Clusters, II edizione*

Anche se la prima definizione di cluster si trova solo a pagina 72 ("un gruppo di calcolatori che collaborano"), il testo sembra comprendere tutti i consueti sistemi multiprocessore e multicomputer. Grande attenzione è dedicata allo studio degli aspetti legati ad hardware, software e prestazioni. È bene segnalare che l'originalità dello stile di scrittura dell'autore, a prima vista simpatico e divertente, perde la sua efficacia man mano che si prosegue nella lettura.

Snir et al., *MPI: The Complete Reference Manual*

Il titolo parla da sé: questo è il riferimento per eccellenza per chi voglia imparare a programmare con MPI. Il libro si occupa della comunicazione punto a punto e di gruppo, dei comunicatori, della gestione d'ambiente, del profilo di utilizzo delle risorse e di altro ancora.

Stenstrom et al., "Trends in Shared Memory Multiprocessing"

Benché i multiprocessori con memoria condivisa siano spesso ritenuti supercomputer per il calcolo scientifico intensivo, in realtà quest'attività costituisce solo una piccola frazione del loro mercato. Gli autori dell'articolo discutono la vera composizione del mercato di queste macchine e le implicazioni risultanti sulla loro architettura.

Ungerer et al., "A Survey of Processors with Explicit Multithreading"

L'articolo spiega il funzionamento dei principali tipi di multithreading (a grana fine o grossa e simultaneo) e fornisce numerosi esempi di calcolatori accademici e commerciali che usano questa tecnica.

Wolf, "The Future of Multiprocessor Systems-on-Chips"

La presentazione di tre progetti di sistema nel chip diventa lo spunto per trattare le sfide future poste da hardware e software di tali sistemi. Le questioni di natura hardware comprendono le problematiche legate al tempo reale e alla dissipazione di energia. Quelle di natura software comprendono le problematiche relative al sistema operativo e alla sicurezza.

9.1.9 Aritmetica binaria

Cody, "Analysis of Proposals for the Floating-Point Standard"

Alcuni anni fa, la IEEE ha progettato un'architettura in virgola mobile diventata lo standard de facto per ogni moderna CPU. Cody analizza tutte le proposte e le controversie sorte durante la standardizzazione.

Koren, "Computer Arithmetic Algorithms"

Un libro completo sull'aritmetica che dedica particolare attenzione agli algoritmi veloci per la somma, la moltiplicazione e la divisione. È particolarmente raccomandato a quanti credono di aver imparato alle elementari tutto quanto c'è da sapere sull'aritmetica.

IEEE, *Proc. of the n-th Symposium on Computer Arithmetic*

Diversamente da quanto si crede generalmente, l'aritmetica è un campo di ricerca attivo che produce molti articoli scientifici specializzati. In queste conferenze sono stati presentati i progressi riguardanti somma e moltiplicazione veloci, l'hardware per l'aritmetica VLSI, la tolleranza agli errori, i coprocessori, l'arrotondamento e altro.

Knuth, *Seminumerical Algorithms, III edizione*

Una gran quantità di materiale su sistemi posizionali, aritmetica in virgola mobile, aritmetica a precisione multipla e numeri casuali. Si tratta di materiale che richiede e merita uno studio molto attento.

Wilson, "Floating-Point Survival Kit"

Un'introduzione apprezzabile all'aritmetica in virgola mobile e ai suoi standard, rivolta a coloro che credono che il mondo finisca alla locazione 65.535. Illustra anche alcune applicazioni molto diffuse per il test e confronto di numeri in virgola mobile, tra cui *Linpack*.

9.1.10 Programmazione in linguaggio assemblativo

Blum, *Professional Assembly Language*

Una guida alla programmazione del Pentium in linguaggio assemblativo per professionisti. Dato che la guida si rivolge a un pubblico professionale, dà per scontato che il lettore usi Linux sulla sua macchina e quindi si concentra sull'analisi dell'assemblatore di Linux, degli strumenti di GNU e delle chiamate di sistema di Linux.

Irvine, *Assembly Language for Intel-Based Computers, IV edizione*

La programmazione delle CPU Intel in linguaggio assemblativo costituisce l'oggetto di questo libro. Tra gli altri argomenti trattati ci sono la programmazione di I/O, le macro, i file, il collegamento, gli interrupt e altro ancora.

9.2 Bibliografia alfabetica

Adams, M. e D., Dulchinos, "OpenCable", *IEEE Commun. Magazine*, vol. 39, pp. 98-105, giugno 2001.

Adiga, N.R. et al., "An overview of the BlueGene/L Supercomputer", *Proc. Supercomputing 2002*, ACM, pp. 1-22, 2002.

Adve, S.V. e K., Charachorloo, "Shared memory consistency models: a tutorial", *IEEE Computer Magazine*, vol. 29, pp. 66-76, dicembre 1996.

Adve, S.V. e M., Hill, "Weak Ordering: a new definition", *Proc. 17th Ann. Int'l Symp. on Computer Arch.*, ACM, pp. 2-14, 1990.

Agerwala, T. e J., Cocke, "High performance reduced instruction set processors", IBM T.J. Watson Research Center Technical Report RC12434, 1987.

Alameldeen, A.R. e D.A., Wood, "Adaptive cache compression for high-performance processors", *Proc. 31st Ann. Int'l Sym. on Computer Arch.*, ACM, pp. 212-223, 2004.

Almasi, G.S. et al., "System management in the BlueGene/L Supercomputer", *Proc. 17th Int'l Parallel and Distr. Proc. Symp.*, IEEE, 2003a.

Almasi, G.S. et al., "An overview of the Bluegene/L System Software Organization", *Par. Proc. Letters*, vol. 13, pp. 561-574, aprile 2003b.

Amza, C., A., Cox, S., Dwarkadas et al., "TreadMarks: shared memory computing on a network of workstations", *IEEE Computer Magazine*, vol. 29, pp. 18-28, febbraio 1996.

Anderson, D., *Universal serial bus system architecture*, Reading, MA, Addison-Wesley, 1997.

Anderson, D., R., Budruk e T., Shanley, *PCI express system architecture*, Reading, MA, Addison-Wesley, 2004.

Anderson, T.E., D.E., Culler e D.A., Patterson et al., "A case for NOW (Networks of workstations)", *IEEE Micro Magazine*, vol. 15, pp. 54-64, gennaio 1995.

Antonakos, J.L., *The Pentium microprocessor*, Upper Saddle River, NJ, Prentice Hall, 1997.

August, D.I., D.A., Connors, S.A., Mshlke et al., "Integrated predicated and speculative execution in the IMPACT EPIC Architecture", *Proc. 25th Ann. Int'l Symp. on Computer Arch.*, ACM, pp. 227-237, 1998.

Ayala, K., *The 8051 microcontroller*, III ed., Clifton Park, NY, Thomson Delmar Learning, 2004.

Bal, H.E., *Programming distributed systems*, Hemel Hempstead, England, Prentice Hall Int'l, 1991.

Bal, H.E., R., Bhoedjang, R., Hofman et al., "Performance evaluation of the Orca Shared Object System", *ACM Trans. on Computer Systems*, vol. 16, pp. 1-40, gennaio-febbraio 1998.

Bal, H.E., M.F., Kaashoek e A.S., Tanenbaum, "Orca: a language for parallel programming of distributed systems", *IEEE Trans. on Software Engineering*, vol. 18, pp. 190-205, marzo 1992.

Bal, H.E. e A.S., Tanenbaum, "Distributed programming with shared data", *Proc. 1988 Int'l Conf. on Computer Languages*, IEEE, pp. 82-91, 1988.

Barroso, L.A., J., Dean e U., Holzle, "Web search for a planet: the Google Cluster Architecture", *IEEE Micro Magazine*, vol. 23, pp. 22-28, marzo-aprile 2003.

Bechini, A., T.M., Conte e C.A., Prete, "Opportunities and challenges in embedded systems", *IEEE Micro Magazine*, vol. 24, pp. 8-9, luglio-agosto 2004.

Benini, L. e G., De Micheli, "Networks on chips" a new SoC Paradigm", *IEEE Computer Magazine*, vol. 35, pp. 70-78, gennaio 2002.

Berman, F., G., Fox e A.J.G., Hey, "Grid computing: making the global infrasttructure a reality", Hoboken, NJ, John Wiley, 2003.

Bjornson, R.D., "Linda on distributed memory multiprocessors", Ph.D. Thesis, Yale Univ., 1993.

Blum, R., *Professional Assembly Language*, New York, Wiley, 2005.

Blumrich, M., D., Chen, G., Chiu et al., "An overview of the BlueGene/L System", *IBM J. Research and Devel.*, vol. 49, marzo-maggio 2005.

Borkar, S., "Getting gigascale chips", *Queue*, pp. 26-33, ottobre 2003.

Bose, P., "Computer architecture research: shifting priorities and newer challenger", *IEEE Micro Magazine*, vol. 24, p. 5, novembre-dicembre 2004.

Bouknight, W.J., S.A., Denenberg, D.E., McIntyre et al., "The Illiac IV System", *Proc. IEEE*, pp. 369-388, aprile 1972.

Brightwell, R., W., Camp, B., Cole et al., "Architectural specification for massively parallel computers. An experience and measurement-based approach", *Concurrency and computation: practice and experience*, vol. 17, pp. 1-46, 2005.

Bryant, R.E. e D., O'Hallaron, *Computer systems a programmer's perspective*, Upper Saddle River, NJ, Prentice Hall, 2003.

Buchanan, W. e A., Wilson, *Advanced PC architecture*, Reading, MA: Addison-Wesley, 2001.

Burger, D. e J.R., Goodman, "Billion-Transistor architectures: there and back again", *IEEE Computer Magazine*, vol. 37, pp. 22-28, marzo 2004.

Burkhardt, H., S., Frank, B., Knobe et al., "Overview of the KSR-1 Computer System", Technical Report KSR-TR-9202001, Kendall Square Research Corp, Cambridge, MA, 1992.

Cain, H. e M., Lipasti, "Memory ordering: a value-based approach", *Proc. 31th Ann. Int'l Symp. on Computer Arch.*, ACM, pp. 90-101, 2004.

Calcutt, D., F., Cowan e H., Parchizadeh, *8051 Microcontrollers: an applications based introduction*, Oxford, Newnes, 2004.

Carriero, N. e D., Gelernter, "Linda in context", *Commun. of the ACM*, vol. 32, pp. 444-458, aprile 1989.

Charlesworth, A., "The Sun fireplane interconnect", *IEEE Micro Magazine*, vol. 22, pp. 36-45, gennaio-febbraio 2002.

Charlesworth, A., "The Sun fireplane interconnect", *Proc. Conf. on High Perf. Networking and Computing*, ACM, 2001.

Charlesworth, A., A., Phelps, R., Williams et al., "Gigaplane-XB: extending the ultra enterprise family", *Proc. Hot Interconnects V*, IEEE, 1998.

Chen, L., S., Dropsho e D.H., Albonesi, "Dynamic data dependence tracking and its application to branch prediction", *Proc. Ninth Int'l Symp. on High-Performance Computer Arch.*, IEEE, pp. 65-78, 2003.

Chou, Y., B., Fahs e S., Abraham, "Microarchitecture optimizations for exploiting Memory-level parallelism", *Proc. 31stAnn. Int'l Symp. on Computer Arch.*, ACM, pp. 76-77, 2004.

Claasen, T.A.C.M., "System on a chip: changing IC design today and in the future", *IEEE Micro Magazine*, vol. 23, pp. 20-26, maggio-giugno 2003.

Cody, W.J., "Analysis of proposals for the floating-point standard", *IEEE Computer Magazine*, vol. 14, pp. 63-68, marzo 1981.

Cohen, D., "On holy wars and a plea for peace", *IEEE Computer Magazine*, vol. 14, pp. 48-54, otttobre 1981.

Colwell, R., *The Pentium chronicles*, New York, Wiley, 2005.

Comer, D.E., "Network systems design using network processors: agere version", Upper Saddle River, NJ, Prentice Hall, 2005.

Corbato, F.J., "PL/1 as a tool for system programming", *Datamation*, vol. 15, pp. 68-76, maggio 1969.

Corbato, F.J. e V.A., Vyssotsky, "Introduction and overview of the MULTICS System", *Proc. FJCC*, pp. 185-196, 1965.

Crowley, P., M.A., Franklin, H., Hadimioglu et al., *Network processor design: issues and practices*, vol. 1, San Francisco, Morgan Kaufmann, 2002.

Dally, W.J. e B.P., Towles, *Principles and practices of interconnection networks*, San Francisco, Morgan Kaufmann, 2004.

Daneshbeh, A.K. e M.A., Hasan, "Area efficient high speed elliptic curve cryptoprocessor for random curves", *Proc. Int'l Conf. on Inf. Tech.: Codingspeculative excution and Computing*, IEEE, pp. 588-593, 2004.

Dean, A.G., "Efficient real-time fine-grained concurrency on low-cost microcontrollers", *IEEE Micro Magazine*, vol. 24, pp. 10-22, luglio-agosto 2004.

Denning, P.J., "The working set model for program behavior", *Commun. of the ACM*, vol. 11, pp. 323-333, maggio 1968.

Dijkstra, E.W., "GOTO statement considered harmful", *Commun. of the ACM*, vol. 11, pp.147-148, marzo 1968a.

Dijkstra, E.W., "Co-operating sequential processes", *Programming Languages*, F. Genuys (ed.), New York, Academic Press, 1968b.

Donaldson, G. e D., Jones, "Cable television broadband network architectures", *IEEE Commun. Magazine*, vol. 39, pp. 122-126, giugno 2001.

Dongarra, J., I., Foster, G., Fox et al., *The sourcebook of parallel computing*, San Francisco, Morgan Kaufman, 2003.

Dubois, M., C., Scheurich e F.A., Briggs, "Memory access buffering in multiprocessors", *Proc. 13th Ann. Int'l Symp. on Computer Arch.*, ACM, pp. 434-442, 1986.

Dulong, C., "The IA-64 architecture at work", *IEEE Computer Magazine*, vol. 31, pp. 24-32, luglio 1998.

Dutta-Roy, A., "An overview of cable modem technology and market perspectives", *IEEE Commun. Magazine*, vol. 39, pp. 81-88, giugno 2001.

Eschmann, F., B., Klauer, R., Moore et al., "SDAARC: an extended cache-only memory architecture", *IEEE Micro Magazine*, vol. 22, pp. 62-70, maggio-giugno 2002.

Faggin, F., M.E., Hoff, S., Jr. Mazor et al., "The history of the 4004", *IEEE Micro Magazine*, vol. 16, pp. 10-20, novembre 1996.

Falcon, A., J., Stark, A., Ramirez et al., "Prophet/critic hybrid branch prediction", *Proc. 31th Ann. Int'l Symp. on Computer Arch.*, ACM, pp. 250-261, 2004.

Fisher, J.A. e S.M., Freudenberger, "Predicting conditional branch directions from previous runs of a program", *Proc. Fifth Int'l Conf. on Arch. Support for Prog. Lang. and Operating Syst.*, ACM, pp. 85-95, 1992.

Floyd, T.L., *Digital fundamentals*, VIII ed., Upper Saddle River, NJ, Prentice Hall, 2002.

Flynn, D., "AMBA: enabling reusable on-chip designs", *IEEE Micro Magazine*, vol. 17, pp. 20-27, luglio 1997.

Flynn, M.J., "Some computer organizations and their effectiveness", *IEEE Trans. on Computers*, vol. C-21, pp. 948-960, settembre 1972.

Foster, I. e C., Kesselman, *The grid 2: blueprint for a new computing infrastructure*, San Francisco, Morgan Kaufman, 2003.

Foster, I., C., Kesselman, J.M., Nick et al., "Grid services for distributed systems integration", *IEEE Computer Magazine*, vol. 35, pp. 37-46, giugno 2002.

Foster, I. e C., Kesselman, "Globus: a metacomputing infrastructure toolkit", *Int'l J. of Supercomputer Applications*, vol. 11, pp. 115-128, 1998a.

Foster, I. e C., Kesselman, "The globus project: a status report", *IPPS/SPDP '98 Heterogeneous Computing Workshop*, IEEE, pp. 4-18, 1998b.

Fotheringham, J., "Dynamic storage allocation in the atlas computer including an automatic use of a backing store", *Commun. of the ACM*, vol. 4, pp. 435-436, ottobre 1961.

Geist, A., A., Beguelin, J., Dongarra et al., *PVM: Parallel Virtual Machine - A user's guide and tutorial for networked parallel computing*, Cambridge, MA, MIT Press, 1994.

Gerber, R. e A., Binstock, *Programming with hyper-threading technology*, Santa Clara, CA, Intel Press, 2004.

Ghemawat, S., H., Gobioff e S.-T., Leung, "The google file system", *Proc. 19th Symp. on Operating Systems Principles*, ACM, pp. 29-43, 2003.

Goodman, J.R., "Using cache memory to reduce processor memory traffic", *Proc. 10th Ann. Int'l Symp. on Computer Arch.*, ACM, pp. 124-131, 1983.

Goodman, J.R., "Cache consistency and sequential consistency", Tech. Rep. 61, IEEE Scalable Coherent Interface Working Group, IEEE, 1989.

Grimshaw, A.S. e W., Wulf, "Legion: a view from 50.000 feet", *Proc. Fifth Int'l Symp. on High-Performance Distributed Computing*, IEEE, pp. 89-99, agosto 1996.

Grimshaw, A.S. e W., Wulf, "The legion vision of a worldwide virtual computer", *Commun. of the ACM*, vol. 40, pp. 39-45, gennaio 1997.

Gropp, W., E., Lusk e A., Skjellum, "Using MPI: portable parallel programming with the message passing interface", Cambridge, MA, MIT Press, 1994.

Gurumurthi, S., M., Sivasubramaniam, Kandemir et al., "Reducing disk power consumption in servers with DRPM", *IEEE Computer Magazine*, vol. 36, pp. 59-66, dicembre 2003.

Hagersten, E., A., Landin e S., Haridi, "DDM - A cache-only memory architecture", *IEEE Computer Magazine*, vol. 25, pp. 44-54, settembre 1992.

Hamacher, V.V., Z.G., Vranesic e S.G., Zaky, *Computer Organization*, V ed., New York, McGraw-Hill, 2001.

Hamming, R.W., "Error detecting and error correcting codes", *Bell Syst. Tech. J.*, vol. 29, pp. 147-160, aprile 1950.

Hammond, L., V., Wong, M., Chen et al., "Transactional memory coherence and consistency", *Proc. 31th Ann. Int'l Symp. on Computer Arch.*, ACM, pp. 102-113, 2004.

Handy, J., *The cache memory book*, II ed., Orlando, FL, Academic Press, 1998.

Hart, J.M., *Win32 system programming*, Reading, MA, Addison-Wesley, 1997.

Heath, S., *Embedded systems design*, Oxford, Newnes, 2003.

Henkel, J., X.S., Hu e S.S., Bhattacharyya, "Taking on the embedded system challenge", *IEEE Computer Magazine*, vol. 36, pp. 35-37, aprile 2003.

Hennessy, J.L., "VLSI processor architecture", *IEEE Trans. on Computers*, vol. C-33, pp. 1221-1246, dicembre 1984.

Hennessy, J.L. e D.A., Patterson, *Computer architecture. A quantitative approach*, III ed., San Francisco, Morgan Kaufmann, 2003.

Hill, M., "Multiprocessors should support simple memory-consistency models", *IEEE Computer Magazine*, vol. 31, pp. 28-34, agosto 1998.

Hinton, G., D., Sager, M., Upton et al., "The microarchtecture of the Pentium 4", *Intel Technology Journal*, vol. 5, pp. 1-12, gennaio-marzo 2001.

Hoare, C.A.R., "Monitors, an operating system structuring concept", *Commun. of the ACM*, vol. 17, pp. 549-557, ottobre 1974 (Erratum in *Commun. of the ACM*, vol. 18, p. 95, febbraio 1975).

Huh, J., D., Burger, J., Chang et al., "Speculative incoherent cache protocols", *IEEE Micro Magazine*, vol. 24, pp. 104-109, novembre-dicembre 2004.

Hwang, K. e Z., Xu, *Scalable parallel computing*, New York, McGraw-Hill, 1998.

Hwu, W.-M., "Introduction to predicated execution", *IEEE Computer Magazine*, vol. 31, pp. 49-50, gennaio 1998.

Irvine, K., *Assembly language for intel-based computers*, IV ed., Upper Saddle River, NJ, Prentice Hall, 2002.

Jacob, B. e T., Mudge, "Virtual memory: issues of implementation", *IEEE Computer Magazine*, vol. 31, pp. 33-43, giugno 1998a.

Jacob, B. e T., Mudge, "Virtual memory in contemporary microprocessors", *IEEE Micro Magazine*, vol. 18, pp. 60-75, luglio-agosto 1998b.

Jerraya, A.A. e W., Wolf, *Multiprocessor Systems-on-a-Chip*, San Francisco, Morgan Kaufmann, 2005.

Jimenez, D.A., "Fast path-based neural branch prediction", *Proc. 36th Int'l Symp. on Microarchitecture*, IEEE, pp. 243-252, 2003.

Johnson, K.L., M.F., Kaashoek e D.A., Wallach, "CRL: high-performance all-software distributed shared memory", *Proc. 15th Symp. on Operating Systems Principles*, ACM, pp. 213-228, 1995.

Johnson, M., *Superscalar Microprocessor Design*, Upper Saddle River, NJ, Prentice Hall, 1991.

Kalla, R., B., Sinharoy e J.M., Tendler, "IBM Power5 Chip: a dual-core multithreaded processor", *IEEE Micro Magazine*, vol. 24, pp. 40-47, marzo-aprile 2004.

Kapasi, U.J., S., Rixner, W.J., Dally et al., "Programmable stream processors", *IEEE Computer Magazine*, vol. 36, pp. 54-62, agosto 2003.

Kapil, S., H., McGhan e J., Lawrendra, "A chip multithreaded processor for network-facing workloads", *IEEE Micro Magazine*, vol. 24, pp. 20-30, marzo-aprile 2004.

Katz, R.H. e G., Borriello, *Contemporary Logic Design*, Upper Saddle River, NJ, Prentice Hall, 2004.

Kaufman, C., R., Perlman e M., Speciner, *Network Security*, II ed., Upper Saddle River, NJ, Prentice Hall, 2002.

Kermarrec, A.-M., I., Kuz, M., Van Steen et al., "A framework for consistent replicated web objects", *Proc. 18th Int'l Conf. on Distr. Computing Syst.*, IEEE, pp. 276-284, 1998.

Kim, N.S., T., Austin, D., Blaauw et al., "Leakage current: Moore's Law meets static power", *IEEE Computer Magazine*, vol. 36, pp. 68-75, dicembre 2003.

Knuth, D.E., "An empirical study of FORTRAN programs", *Software-Practice & Experience*, vol. 1, pp. 105-133, 1971.

Knuth, D.E., *The art of computer programming: fundamental algorithms*, III ed., Reading, MA, Addison-Wesley, 1997.

Knuth, D.E., *The art of computer programming: seminumerical algorithms*, III ed., Reading, MA, Addison-Wesley, 1998.

Kogel, T. e H., Myer, "Heterogeneous MP-SoC: the solution to energy-efficient signal processing", *Proc. 41st Ann. Conf. on Design Automation*, IEEE, pp. 686-691, 2004.

Kontothanassis, L., G., Hunt, R., Stets et al., *VM-based shared memory on low latency remote memory access networks, Proc. 24th Ann. Int'l Symp. on Computer Arch.*, ACM, pp. 157-169, 1997.

Koren, I., *Computer arithmetic algorithms*, Natick, MA, A.K. Peters, 2002.

Koufaty, D. e D.T., Marr, "Hyperthreading technology in the netburst microarchitecture", *IEEE Micro Magazine*, vol. 23, pp. 56-65, marzo-aprile 2003.

Kumar, R., N.P., Jouppi e D.M., Tullsen, "Conjoined-core chip multiprocessing", *Proc. 37th Int'l Symp. on Microarchitecture*, IEEE, pp. 195-206, 2004.

Lamport, L., "How to make a multiprocessor computer that correctly executes multiprocess programs", *IEEE Trans. on Computers*, vol. C-28, pp. 690-691, settembre 1979.

Larowe, R.P. e C.S., Ellis, "Experimental comparison of memory management policies for NUMA multiprocessors", *ACM Trans. on Computer Systems*, vol. 9, pp. 319-363, novembre 1991.

Lavagno, L., "Systems on a chip: the next electronic frontier", *IEEE Micro Magazine*, vol. 22, pp. 14-15, settembre-ottobre 2002.

Lawton, G., "Will network processor units live up to their promise?", *IEEE Computer Magazine*, vol. 37, pp. 13-15, aprile 2004.

Lekkas, P.C., *Network processors: architectures, protocols and platforms*, New York, McGraw-Hill, 2003.

Levine, J.R., *Linkersand loaders*, San Francisco, Morgan Kaufmann, 2000.

Li, K. e P., Hudak, "Memory coherence in shared virtual memory systems", *ACM Trans. on Computer Systems*, vol. 7, pp. 321-359, novembre 1989.

Lima, F., L., Carro, R., Velazco et al., "Injecting multiple upsets in a SEU Tolerant 8051 Micro-Controller", *Proc. Eighth IEEE Int'l On-Line Testing Workshop IEEE*, p. 194, luglio 2002.

Lines, A., "Asynchronous interconnect for synchronous SoC design", *IEEE Micro Magazine*, vol. 24, pp. 32-41, gennaio-febbraio 2004.

Lu, H., A.L., Cox, S., Dwarkadas et al., "Software distributed shared memory support for irregular applications", *Proc. Sixth Conf. on Prin. and Practice of Parallel Progr.*, pp. 48-56, giugno 1997.

Lukasiewicz, J., *Aristotle's syllogistic*, II ed., Oxford, Oxford University Press, 1958.

Lutz, J. e A., Hasan, "High performance FPGA based elliptic curve cryptographic Co-Processor", *Proc. Int'l Conf. on Inf. Tech.: Coding and Computing*, IEEE, pp. 486-492, 2004.

Lyytinen, K. e Y., Yoo, "Issues and challenges in ubiquitous computing", *Commun. of the ACM*, vol. 45, pp. 63-65, dicembre 2002.

Mackenzie, I.S. e R., Phan, *The 8051 Microcontroller*, IV ed., Upper Saddle River, NJ, Prentice Hall, 2005.

Mano, M.M. e C.R., Kime, *Logic and computer design fundamentals*, III ed., Upper Saddle River, NJ, Prentice Hall, 2003.

Martin, A.J., M., Nystrom, K., Papadantonakis et al., "The lutonium: a sub-nanojoule asynchronous 8051 microcontroller", *Proc. Ninth Int'l Symp. on Asynchronous Circuits and Systems IEEE*, pp. 14-23, 2003.

Martin, R.P., A.M., Vahdat, A.M., Culler et al., "Effects of communication latency, overhead and bandwidth in a cluster architecture", *Proc. 24th Ann. Int'l Symp. on Computer Arch.*, ACM, pp. 85-97, 1997.

Mayhew, D. e V., Krishnan, "PCI express and advanced switching: evolutionary path to building next generation interconnects", *Proc. 11th Symp. on High Perf. Interconnects IEEE*, pp. 21-29, agosto 2003.

Mazidi, M.A., McKinlay e J.G., Mazidi, *8051 Microcontroller and embedded systems*, Upper Saddle River, NJ, Prentice Hall, 2005.

Mazidi, M.A. e J.G., Mazidi, *The 80x86 IBM PC and compatible computers*, IV ed., Upper Saddle River, NJ, Prentice Hall, 2002.

McKnight, L.W., J., Howison e S., Bradner, "Wireless grids", *IEEE Internet Computing*, vol. 8, pp. 24-31, luglio-agosto 2004.

McKusick, M.K., K., Bostic, M., Karels et al., "The design and implementation of the 4.4 BSD Operating System", Reading, MA, Addison-Wesley, 1996.

McKusick, M.K., W.N., Joy, S.J., Leffler et al., "A fast file system for UNIX", *ACM Trans. on Computer Systems*, vol. 2, pp. 181-197, agosto 1984.

McNairy, C. e D., Soltis, "Itanium 2 processor microarchitecture", *IEEE Micro Magazine*, vol. 23, pp. 44-55, marzo-aprile 2003.

Min, R., W.-Ben., Jone e Y., Hu, "Location cache: a low-power L2 cache system", *Proce. 2004 Int'l Symp. on Low Power Electronics and Design*, IEEE, pp. 120-125, agosto 2004.

Messmer, H.-P., *The indispensible PC hardware book*, IV ed., Reading, MA, Addison-Wesley, 2001.

Moudgill, M. e S., Vassiliadis, "Precise interruptus", *IEEE Micro Magazine*, vol. 16, pp. 58-67, gennaio 1996.

Mullender, S.J. e A.S., Tanenbaum, "Immediate files", *Software-Practice and Experience*, vol. 14, pp. 365-368, 1984.

Nesbit, K.J. e J.E., Smith, "Data cache prefetching using a global history buffer", *Proc. 10th Int'l Symp. on High Perf. Computer Arch.*, IEEE, pp. 96-106, 2004.

Ng, S.W., "Advances in disk technology: performance issues", *IEEE Computer Magazine*, vol. 31, pp. 75-81, maggio 1998.

Nickolls, J., L.J., Madar III, S., Johnson et al., "Calisto: a low-power single-chip multiprocessor communications platform", *IEEE Micro Magazine*, vol. 23, pp. 29-43, marzo 2003.

Norton, P. e J., Goodman, *Inside the PC*, VIII ed., Indianapolis, IN, Sams, 1999.

Null, L. e J., Lobur, *The essentials of computer organization and architecture*, Sudbury, MA, Jones and Bartlett, 2003.

O'Connor, J.M. e M., Tremblay, "PicoJava-I: the Java virtual machine in Hardware", *IEEE Micro Magazine*, vol. 17, pp. 45-53, marzo-aprile 1997.

Organick, E., *The MULTICS system*, Cambridge, MA, MIT Press, 1972.

Oskin, M., F.T., Chong e I.L., Chuang, "A practical architecture for reliable quantum computers", *IEEE Computer Magazine*, vol. 35, pp. 79-87, gennaio 2002.

Ouadjaout, S. e D., Houzet, "Easy SoC design with VCI SystemC Adapters", *Proc. Digital System Design*, IEEE, pp. 316-323, 2004.

Papaefstathiou, I., N.A., Nikolaou, B., Doshi et al., "Network processors for future high-end systems and applications", *IEEE Micro Magazine*, vol. 24, pp. 7-9, settembre-ottobre 2004.

Papamarcos, M. e J., Patel, "A low overhead coherence solution for multiprocessors with private cache memories", *Proc. 11th Ann. Int'l Symp. on Computer Arch.*, ACM, pp. 348-354, 1984.

Parikh, D., K., Skadron, Y., Zhang et al., "Power-Aware Branch Prediction: characterization and design", *IEEE Trans. on Computers*, vol. 53, pp. 168-186, febbraio 2004.

Patterson, D.A., "Reduced instruction set computers", *Commun. of the ACM*, vol. 28, pp. 8-21, gennaio 1985.

Patterson, D.A., G., Gibson e R., Katz, "A case for redundant arrays of inexpensive disks (RAID)", *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, ACM, pp. 109-166, 1988.

Patterson, D.A. e J.L., Hennessy, *Computer organization and design*, III ed., San Francisco, Morgan Kaufmann, 2005.

Patterson, D.A. e C.H., Sequin, "A VLSI RISC", *IEEE Computer Magazine*, vol. 15, pp. 8-22, settembre 1982.

Paul, R.P., *SPARC architecture, assembly language, programming and C*, Upper Saddle River, NJ, Prentice Hall, 1994.

Pfister, G.F., *In search of clusters*, II ed., Upper Saddle River, NJ, Prentice Hall, 1998.

Popescu, B.C., M. Van, Steen e A.S., Tanenbaum, "A security architecture for object-based distributed systems", *Proc. 18th Annual Computer Security Appl. Conf.*, ACM, pp. 161-171, 2002.

Pountain, D., "Pentium: more RISC than CISC", *Byte*, vol. 18, pp. 195-204, settembre 1993.

Price, D., "A history of calculating machines", *IEEE Micro Magazine*, vol. 4, pp. 22-52, gennaio 1984.

Radin, G., "The 801 minicomputer", *Computer Arch. News*, vol. 10, pp. 39-47, marzo 1982.

Raman, S.K., V., Pentkovski e J., Keshava, "Implementing streaming SIMD extensions on the Pentium III Processor", *IEEE Micro Magazine*, vol. 20, pp. 47-57, luglio-agosto 2000.

Ravikumar, C.P., "Multiprocessor architectures for embedded system-on-a-chip Applications", *Proc. 17th Int'l Conf. on VLSI Design*, IEEE, pp. 512-519, gennaio 2004.

Ritchie, D.M. e K., Thompson, "The UNIX time-sharing system", *Commun. of the ACM*, vol. 17, pp. 365-375, luglio 1974.

Robinson, G.S., "Toward the age of smarter storage", *IEEE Computer Magazine*, vol. 35, pp. 35-41, dicembre 2002.

Rosenblum, M. e J.K., Ousterhout, "The design and implementation of a log-structured file system", *Proc. Thirteenth Symp. on Operating System Principles*, ACM, pp. 1-15, 1991.

Roth, C.H., *Fundamentals of logic design*, V ed., Florence, KY, Thomson Engineering, 2003.

Russinovich, M.E. e D.A., Solomon, *Microsoft Windows internals*, IV ed., Redmond, WA, Microsoft Press, 2005.

Rusu, S., H., Muljono e B., Cherkauer, "Itanium 2 Processor 6M", *IEEE Micro Magazine*, vol. 24, pp. 10-18, marzo-aprile 2004.

Saha, D. e A., Mukherjee, "Pervasive computing: a paradigm for the 21st century", *IEEE Computer Magazine*, vol. 36, pp. 25-31, marzo 2003.

Sakamura, K., "Making computers invisibile", *IEEE Micro Magazine*, vol. 22, p. 711, 2002.

Saloman, D., *Assemblers and loaders*, Upper Saddle River, NJ, Prentice Hall, 1993.

Scales, D.J., K., Gharachorloo e C.A., Thekkath, "Shasta: a low-overhead software-only approach for supporting fine-grain shared memory", *Proc. Seventh Int'l Conf. on Arch. Support for Prog. Lang. and Oper. Syst.*, ACM, pp. 174-185, 1996.

Scheible, J.P., "A survey of storage options", *IEEE Computer Magazine*, vol. 35, pp. 42-46, dicembre 2002.

Seltzer, M., K., Bostic, M.K., McKusick et al., "An implementation of a log-structured file system for UNIX", *Proc. Winter 1993 USENIX Technical Conf.*, pp. 307-326, 1993.

Shanley, T. e D., Anderson, *PCI system architecture*, IV ed., Reading, MA, Addison-Wesley, 1999.

Shriver, B. e B., Smith, *The anatomy of a high-performance microprocessor: a systems perspective*, Los Alamitos, CA, IEEE Computer Society, 1998.

Sima, D., "Superscalar instruction issue", *IEEE Micro Magazine*, vol. 17, pp. 28-39, settembre-ottobre 1997.

Sima, D., T., Fountain e P., Kacsuk, *Advanced computer architectures: a design space approach*, Reading, MA, Addison-Wesley, 1997.

Slater, R., *Portraits in silicon*, Cambridge, MA, MIT Press, 1987.

Sohi, G.S. e A., Roth, "Speculative multithreaded processors", *IEEE Computer Magazine*, vol. 34, pp. 66-73, aprile 2001.

Snir, M., S.W., Otto, S., Huss-Lederman et al., *MPI: the complete reference manual*, Cambridge, MA, MIT Press, 1996.

Solari, E. e B., Congdon, *PCI express design & system architecture*, Research Tech, INc., 2005.

Solari, E. e G., Willse, *PCI and PCI-X hardware and software*, VI ed., San Diego, CA, Annabooks, 2004.

Stallings, W., *Computer organization and architecture*, VI ed., Upper Saddle River, NJ, Prentice Hall, 2003 (trad. it. a cura di O. D'Antona, *Architettura e organizzazione dei calcolatori*, VI ed., Pearson Education Italia, 2004).

Stenstrom, P., E., Hagersten, D.J., Lilja et al., "Trends in shared memory multiprocessing", *IEEE Computer Magazine*, vol. 30, pp. 44-50, dicembre 1997.

Stets, R., S., Dwarkadas, N., Hardavellas et al., "CASHMERE-2L: software coherent shared memory on clustered remote-write networks", *Proc. 16th Symp. on Operating Systems Principles*, ACM, pp. 170-183, 1997.

Suh, T., H.-H.S., Lee, D.M., Blough, "Integrating cache coherence protocols for heterogeneous multiprocessor systems – Part 1", *IEEE Micro Magazine*, vol. 24, pp. 33-41, luglio 2004.

Summers, C.K., *ADSL: standards, implementation and architecture*, Boca Raton, FL, CRC Press, 1999.

Sunderram, V.B., "PVM: a framework for parallel distributed computing", *Concurrency: practice and experience*, vol. 2, pp. 315-339, dicembre 1990.

Swan, R.J., S.H., Fuller e D.P., Siewiorek, "Cm*-A modular multiprocessor", *Proc. NCC*, pp. 645-655, 1977.

Tan, W.M., *Developing USB PC Peripherals*, San Diego, CA, Annabooks, 1997.

Tanenbaum, A.S., "Computer networks", *Upper Saddle River*, NJ, Prentice Hall, 2003 (trad. it. *Reti di calcolatori*, IV ed., Pearson Education Italia, 2003).

Tanenbaum, A.S., "Implications of structured programming for machine architecture", *Commun. of the ACM*, vol. 21, pp. 237-246, marzo 1978.

Tanenbaum, A.S., *Operating systems: design and implementation*, Upper Saddle River, NJ, Prentice Hall, 1987.

Tanenbaum, A.S. e A.W., Woodhull, *Operating systems: design and implementation*, II ed., Upper Saddle River, NJ, Prentice Hall, 1997.

Thompson, K., "UNIX implementation", *Bell Syst. Tech. J.*, vol. 57, pp. 1931-1946, luglio-agosto 1978.

Treleaven, P., "Control-Driven, Data-Driven, and Demand-Driven computer architecture", *Parallel Computing*, vol. 2, 1985.

Tremblay, M. e J.M., O'Connor, "UltraSPARC I: A Four-Issue Processor Supporting Multimedia", *IEEE Micro Magazine*, vol. 16, pp. 42-50, marzo 1996.

Triebel, W.A., *The 80386, 80486, and Pentium Processor*, Upper Saddle River, NJ, Prentice Hall, 1998.

Tuck, N. e D.M., Tullsen, "Initial observations of the simultaneous multithreading Pentium 4 Processor", *Proc. 12th Int'l Conf. on Parallel Arch. and Compilation Techniques*, IEEE, pp. 26-35, 2003.

Unger, S.H., "A computer oriented toward spatial problem", *Proc. IRE*, vol. 46, pp. 1744-1750, 1958.

Vahalia, U., *UNIX Internals*, Upper Saddle River, NJ, Prentice Hall, 1996.

Vahid, F., "The softening of hardware", *IEEE Computer Magazine*, vol. 36, pp. 27-34, aprile 2003.

Van Steen, M., P.C., Homburg e A.S., Tanenbaum, "The architectural design of globe: a wide-area distributed system", *IEEE Concurrency*, vol. 7, pp. 70-78, gennaio-marzo 1999.

Vetter, P., D., Goderis, L., Verpooten et al., "Systems aspects of APON/VDSL deployment", *IEEE Commun. Magazine*, vol. 38, pp. 66-72, maggio 2000.

Weaver, D.L. e T., Germond, *The SPARC architecture manual, version 9*, Upper Saddle River, NJ, Prentice Hall, 1994.

Weiser, M., "The computer for the 21st Century", *IEEE Pervasive Computing*, vol. 1, pp. 19-25, gennaio-marzo 2002 (pubblicato in originale su Scientific American, settembre 1991).

Wilkes, M.V., "Computers then and now", *J. ACM*, vol. 15, pp. 1-7, gennaio 1968.

Wilkes, M.V., "The best way to design an automatic calculating machine", *Proc. Manchester Univ. Computer Inaugural Conf.*, 1951.

Wilson, J., "Challenges and trends in processor design", *IEEE Computer Magazine*, vol. 31, pp. 39-48, gennaio 1998.

Wilson, P., "Floating-Point survival kit", *Byte*, vol. 13, pp. 217-226, marzo 1988.

Wolf, W., "The future of multiprocessor systems-on-chips", *Proc. 41st Ann. Conf. on Design Automation*, IEEE, pp. 681-685, 2004.

Appendice A

Aritmetica binaria

L'aritmetica dei calcolatori è un po' diversa da quella che abbiamo studiato alle elementari. La differenza principale sta nel fatto che i calcolatori eseguono operazioni su numeri che hanno una precisione finita e prefissata. Un'altra differenza è che quasi tutti i calcolatori rappresentano i numeri nel sistema binario e non in quello decimale. Questa appendice si occupa precisamente di questi argomenti.

A.1 Numeri a precisione finita

Quando si fanno calcoli aritmetici, in genere non si dà particolare importanza al numero di cifre decimali utilizzate per rappresentare i numeri. I fisici calcolano che nell'universo esistono 10^{78} elettroni, senza preoccuparsi del fatto che ci vogliono 79 cifre decimali per scrivere quel numero per esteso. Una persona che usi carta e penna per calcolare il valore di una funzione e che abbia bisogno di una risposta con sei cifre significative, svolgerà i calcoli mantenendo i risultati intermedi con una precisione di sette, otto cifre, o di quante siano necessarie. Non succede mai che il foglio di carta sia troppo piccolo per contenere i numeri con sette cifre significative.

Le cose sono molto diverse nei calcolatori. Per molti di loro, la quantità di memoria disponibile per la memorizzazione di un numero è fissata al momento della progettazione. Il programmatore può riuscire con qualche sforzo a raddoppiare o a triplicare il numero di cifre che può usare, ma ciò non cambia la natura del problema. La natura intrinsecamente finita di un calcolatore ci costringe a trattare solo numeri rappresentati per mezzo di un numero limitato e costante di cifre, cioè **numeri a precisione finita**.

Per studiare le proprietà dei numeri a precisione finita, cominciamo con l'esaminare l'insieme degli interi positivi rappresentabili con tre cifre decimali, senza virgola e senza segno. Questo insieme ha esattamente 1000 elementi: 000, 001, 002, 003, ..., 999. Con queste limitazioni non possiamo esprimere:

1. i numeri più grandi di 999
2. i numeri negativi
3. le frazioni (proprie)

4. i numeri irrazionali
5. i numeri complessi.

Una proprietà importante dell'aritmetica sull'insieme degli interi è la **chiusura** rispetto alle operazioni di addizione, sottrazione e moltiplicazione. In altre parole, per ogni coppia d'interi i e j, i risultati delle operazioni $i + j$, $i - j$ e $i \times j$ sono tutti interi. L'insieme degli interi non è chiuso rispetto alla divisione, perché esistono valori di i e j per cui i/j non è esprimibile come un intero (per esempio 7/2 e 1/0).

I numeri a precisione finita non sono chiusi rispetto ad alcuna delle quattro operazioni fondamentali, come mostriamo nell'esempio seguente usando numeri decimali con tre cifre decimali:

600 + 600 = 1200 (troppo grande)
003 – 005 = – 2 (negativo)
050 × 050 = 2500 (troppo grande)
007 / 002 = 3,5 (non è intero).

Le violazioni della chiusura possono essere distinte in due classi disgiunte: le operazioni che danno un risultato più grande del massimo dell'insieme (*overflow*) o più piccolo del minimo dell'insieme (*underflow*), e le operazioni il cui risultato non appartiene all'insieme. Le prime tre violazioni dell'esempio sono del primo tipo, la quarta è del secondo.

I calcolatori dispongono di una memoria finita e quindi devono svolgere necessariamente conti aritmetici a precisione finita. Di conseguenza i risultati di alcuni calcoli saranno semplicemente sbagliati dal punto di vista della matematica classica. Un dispositivo di calcolo perfettamente funzionante che fornisca risultati errati può sembrare di primo acchito una stranezza, ma l'errore è una conseguenza logica della sua natura finita. Alcuni calcolatori dispongono di hardware speciale per il rilevamento di errori di overflow.

L'algebra dei numeri a precisione finita è diversa dall'algebra normale. Per esempio consideriamo l'espressione

$$a + (b - c) = (a + b) - c$$

(che è un'identità in base alla proprietà associativa di somma e sottrazione) e valutiamola per $a = 700$, $b = 400$ e $c = 300$. Per calcolare il membro di sinistra cominciamo da $(b - c)$, che dà 100, e aggiungiamo questo valore ad a, per un totale di 800. Per calcolare il secondo membro, calcoliamo prima $(a + b)$, e ciò causa un overflow nell'aritmetica finita degli interi a tre cifre. Il risultato potrebbe dipendere dalla macchina utilizzata per il calcolo, ma non sarà mai 1100. La sottrazione di 300 da un qualunque numero diverso da 1100 non varrà mai 800. L'associatività non vale e l'ordine di esecuzione delle operazioni è discriminante. Come ulteriore esempio, consideriamo la legge distributiva (del prodotto rispetto alla somma):

$$a \times (b - c) = a \times b - a \times c$$

e valutiamo entrambi i membri per $a = 5$, $b = 210$ e $c = 195$. Il primo membro vale 5 × 15, cioè 75, ma il secondo membro non dà 75 perché a × b causa un overflow.

A giudicare da questi esempi, si potrebbe concludere che i calcolatori, pur essendo dei dispositivi per uso generale, sono poco portati per i calcoli aritmetici a causa della loro natura intrinsecamente finita. Ovviamente questa conclusione è errata, ma serve a illustrare l'importanza di comprendere il funzionamento e le limitazioni dei calcolatori.

A.2 Sistemi di numerazione in base fissa

Un numero decimale consiste in una sequenza di cifre decimali più, eventualmente, una virgola decimale. La Figura A.1 ne mostra la forma generale e la sua interpretazione posizionale. La scelta di 10 come **base** per l'elevamento a potenza dipende dal fatto che stiamo usando i decimali, ovvero i numeri in base 10. In informatica conviene spesso usare basi diverse da 10. Le basi più importanti sono 2, 8 e 16 e i sistemi di numerazione basati su di loro si chiamano rispettivamente **binari**, **ottali** ed **esadecimali**.

Un sistema di numerazione in base k richiede k simboli diversi per rappresentare le cifre da 0 a $k - 1$. I numeri decimali si scrivono con le 10 cifre decimali

0 1 2 3 4 5 6 7 8 9

Invece i numeri binari non usano dieci cifre, ma si scrivono con le sole due cifre binarie

0 1

I numeri ottali si costruiscono a partire dalle otto cifre ottali

0 1 2 3 4 5 6 7

mentre i numeri esadecimali richiedono 16 cifre, cioè sei nuovi simboli. Per convenzione si usano le lettere maiuscole da A a F per le cifre che seguono 9. Dunque i numeri esadecimali si scrivono con le cifre

0 1 2 3 4 5 6 7 8 9 A B C D E F

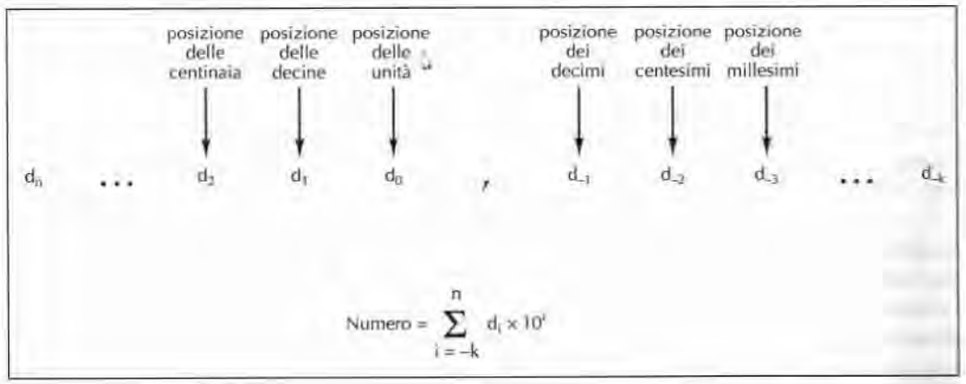


Figura A.1 Forma generale di un numero decimale.

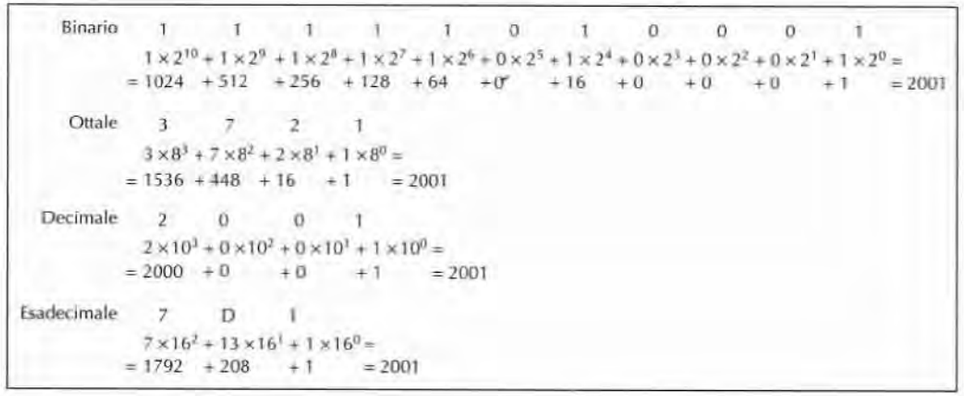


Figura A.2 Il numero 2001 in binario, ottale, decimale ed esadecimale.

Con l'espressione "**bit**" si intende una cifra che può assumere i valori 0 e 1. La Figura A.2 mostra il numero decimale 2001 espresso in binario, in ottale, in decimale e in forma esadecimale. Il numero 7B9 è ovviamente esadecimale, perché il simbolo B si trova solo nei numeri esadecimali. La stringa 111 invece può essere interpretata come un numero in uno qualsiasi dei quattro sistemi di numerazione presentati. Per evitare l'ambiguità, si usa indicare la base con un pedice 2, 8, 10 o 16, tutte le volte che non può essere intuita direttamente dal contesto.

La Figura A.3 mostra un certo numero d'interi non negativi espressi nelle quattro basi. Forse un giorno, fra migliaia di anni, gli archeologi scopriranno questa tabella e la considereranno la Stele di Rosetta dei sistemi di numerazione usati a cavallo tra il tardo XX secolo e gli inizi del XXI secolo.

A.3 Conversione tra basi

La conversione tra i sistemi ottale o esadecimale e binario è facile. Per convertire un numero binario in ottale basta suddividerlo in gruppi di 3 bit, avendo cura di raggruppare i 3 bit appena a sinistra (o a destra) della virgola e poi tutti gli altri bit. Ogni gruppo di 3 bit può essere convertito direttamente in una cifra ottale da 0 a 7, secondo la conversione indicata dalle prime righe della tabella nella Figura A.3. Potrebbe essere necessario aggiungere uno o più zeri in testa o in coda ai gruppi di bit che non hanno 3 cifre.

Anche la conversione da ottale a binario è banale. Ogni cifra ottale viene semplicemente rimpiazzata dall'equivalente numero binario a 3 bit. La conversione da esadecimale a binario è praticamente analoga a quella ottale-binario, con la differenza che le cifre esadecimali corrispondono a gruppi di 4 cifre binarie e non di 3. La Figura A.4 fornisce alcuni esempi di conversioni.

La conversione dei numeri decimali in binario può essere effettuata con due metodi diversi. Il primo è la diretta conseguenza della definizione di numero binario. Si comincia con il sottrarre al numero decimale la più grande potenza di 2 minore del numero stesso,

Decimale	Binario	Ottale	Esadecimale
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	3	3
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10
20	10100	24	14
30	11110	36	1E
40	101000	50	28
50	110010	62	32
60	111100	74	3C
70	1000110	106	46
80	1010000	120	50
90	1011010	132	5A
100	11001000	144	64
1000	1111101000	1750	3E8
2989	101110101101	5655	BAD

Figura A.3 Alcuni numeri decimali e i loro equivalenti binari, ottali, decimali ed esadecimali.

dopodiché si ripete il procedimento con la differenza ottenuta. Una volta scomposto il numero in potenze di 2, il numero binario si ottiene inserendo degli 1 nelle posizioni che corrispondono alle potenze di 2 usate nella scomposizione, lasciando 0 nelle altre posizioni.

L'altro metodo (valido solo per gli interi) consiste nel dividere il numero per 2. Si scrive il quoziente appena sotto il numero originale mentre il resto, 0 o 1, viene scritto accanto al quoziente. A quel punto si ripete il procedimento sul quoziente e si itera finché non si arriva a 0. Il procedimento dà origine a due colonne di numeri, i quozienti e i resti.

Esempio 1

Esadecimale		1		9		4		8	,	B	6
Binario	0	0	0	1	1	0	0	1	.	1	0	1	1	0	1	1	0	0
Ottale		1		4		5		1	0	,	5		5		4			

Esempio 2

Esadecimale		7		B		A		3	,	B	C		4	
Binario	0	1	1	1	1	0	1	1	1	0	1	0	0	0	1	1	,	1	0	1	1	1	0	0	0	1	0	0
Ottale		7		5		6		4		3	,	5		7		0		4										

Figura A.4 Esempi di conversione da ottale a binario e da esadecimale a binario.

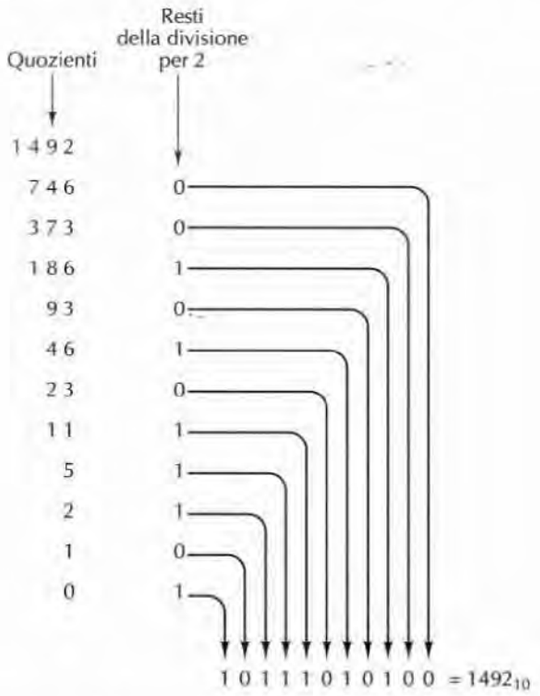


Figura A.5 La conversione del numero decimale 1492 in binario mediante dimezzamenti successivi, partendo dall'alto e procedendo verso il basso. Per esempio, 93 diviso 2 fa 46 con resto 1, riportati nella riga successiva.

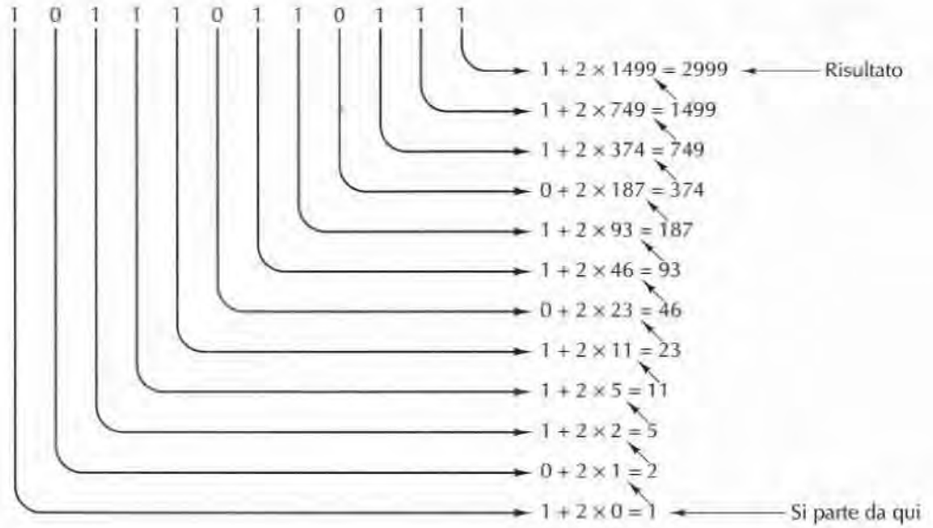


Figura A.6 La conversione del numero binario 101110110111 in decimale mediante raddoppiamenti successivi, a partire dal basso. Ogni riga si ottiene raddoppiando l'elemento della riga precedente e sommandogli il bit corrispondente. Per esempio, 749 è due volte 374 più il bit 1 che si trova in corrispondenza della riga di 749.

Il numero binario si ottiene direttamente scorrendo la colonna dei resti a partire dal basso. La Figura A.5 mostra un esempio di conversione da decimale a binario.

Anche gli interi binari possono essere convertiti in decimale con due metodi. Il primo consiste nel sommare le potenze di 2 che corrispondono alle posizioni degli 1 nel numero binario. Per esempio

$$10110 = 2^4 + 2^2 + 2^1 = 16 + 4 + 2 = 22$$

Nell'altro metodo, il numero binario si scrive verticalmente, un bit per riga, con il bit più significativo nell'ultima riga. Diciamo riga 1 l'ultima riga, riga 2 quella appena sopra e così via. Il numero decimale si costruisce in una colonna parallela a quella del numero binario. Si comincia con lo scrivere 1 nella riga 1. L'elemento della riga n è il doppio della riga $n - 1$ più il bit della riga n (0 o 1). L'elemento della riga più in alto contiene il risultato della conversione. La Figura A.6 illustra un esempio di applicazione di questo metodo.

La conversione da decimale a ottale o esadecimale si può conseguire o passando per la conversione in binario, o mediante sottrazioni successive di potenze di 8 o di 16.

A.4 Numeri binari negativi

Nei calcolatori digitali sono stati usati nel corso degli anni quattro diversi sistemi per la rappresentazione dei numeri negativi. Il primo si chiama **modulo e segno** e utilizza il bit più significativo come bit di segno (0 per il +, e 1 per il –) e i restanti come modulo (valore assoluto) del numero.

Il secondo sistema (che ormai è obsoleto) si chiama **complemento a uno** e anch'esso prevede un bit di segno dove 0 indica i numeri positivi, 1 i negativi. La negazione di un numero si ottiene scambiando tutti i suoi 1 con 0 e viceversa, compreso il bit di segno.

Il terzo sistema si chiama **complemento a due** e prevede un bit di segno, analogo al caso precedente. L'opposto di un numero si ottiene in due passi. Per prima cosa si rimpiazza ogni 1 con uno 0 e viceversa (proprio come nel complemento a uno) e poi si aggiunge 1 al risultato. Il risultato della somma binaria è lo stesso della somma decimale con la differenza che viene generato un resto se la somma è maggiore di 1, non di 9. Per esempio, la conversione di 6 in complemento a due si ottiene con i due passaggi:

00000110 (+6)
11111001 (–6 in complemento a uno)
11111010 (–6 in complemento a due).

L'eventuale resto in corrispondenza del bit più significativo viene ignorato.

Il quarto sistema si chiama **notazione in eccesso di 2^{m-1}** (per numeri di m bit) e rappresenta il numero memorizzando la sua somma con 2^{m-1}. Per esempio, nel caso di numeri di 8 bit ($m = 8$) il sistema si dice essere in eccesso di 128 e memorizza un numero dopo avergli sommato 128. Così –3 diventa $-3 + 128 = 125$ e –3 viene rappresentato dal numero binario di 8 bit che vale 125 (01111101). I numeri da –128 a 127 corrispondono ai numeri da 0 a 255, tutti esprimibili come interi positivi di 8 bit. È abbastanza interessante notare che questo sistema è identico al complemento a due con il bit di segno invertito. La Figura A.7 fornisce alcuni esempi di numeri negativi espressi con i quattro sistemi.

Il modulo con segno e il complemento a uno hanno entrambi due distinte rappresentazioni di zero: uno zero positivo e uno zero negativo. Questa proprietà non è molto gradita. Il sistema in complemento a due non è affetto da questo problema, perché il complemento a due di 0 è ancora 0. Tuttavia anche il sistema in complemento a due ha una sua anomalia: la configurazione formata da un 1 seguito da tutti 0 è il complemento di se stessa. L'effetto di questa proprietà è che l'intervallo dei numeri positivi e quello dei numeri negativi è asimmetrico: esiste un numero negativo che non ha una controparte tra i numeri positivi.

La motivazione che sta dietro a questi problemi è facile da individuare: desideriamo un sistema di codifica che esibisca le due proprietà seguenti:

1. una sola rappresentazione dello zero
2. uguale numero d'interi positivi e interi negativi rappresentati.

N decimale	N binario	–N modulo con segno	–N compl. a 1	–N compl. a 2	–N in eccesso di 128
1	00000001	10000001	11111110	11111111	01111111
2	00000010	10000010	11111101	11111110	01111110
3	00000011	10000011	11111100	11111101	01111101
4	00000100	10000100	11111011	11111100	01111100
5	00000101	10000101	11111010	11111011	01111011
6	00000110	10000110	11111001	11111010	01111010
7	00000111	10000111	11111000	11111001	01111001
8	00001000	10001000	11110111	11111000	01111000
9	00001001	10001001	11110110	11110111	01110111
10	00001010	10001010	11110101	11110110	01110110
20	00010100	10010100	11101011	11101100	01101100
30	00011110	10011110	11100001	11100010	01100010
40	00101000	10101000	11010111	11011000	01011000
50	00110010	10110010	11001101	11001110	01001110
60	00111100	10111100	11000011	11000100	01000100
70	01000110	11000110	10111001	10111010	00111010
80	01010000	11010000	10101111	10110000	00110000
90	01011010	11011010	10100101	10100110	00100110
100	01100100	11100100	10011011	10011100	00011100
127	01111111	11111111	10000000	10000001	00000001
128	Inesistente	Inesistente	Inesistente	10000000	00000000

Figura A.7 Numeri negativi di 8 bit nei quattro sistemi di rappresentazione.

Il fatto è che un insieme di numeri che abbia lo stesso numero di elementi positivi e negativi e una sola rappresentazione di 0 ha necessariamente un numero dispari di elementi, mentre con m bit si può rappresentare un numero pari di stringhe di bit. Perciò, qualunque rappresentazione conterrà sempre una configurazione in più (o in meno) di quanto desiderato. La configurazione eccedente può essere utilizzata per rappresentare –0, per denotare un numero negativo grande (in modulo) o per qualcos'altro; comunque sia resterà sempre una seccatura.

Infine va ricordato che, data una stringa di n bit in complemento a 2, diciamo

$$b_{n-1} \ldots b_1 b_0,$$

il numero da essa rappresentato è ottenuto mediante la formula

$$-b_{n-1} \times 2^{n-1} + \sum_{i=0}^{n-2} b_i \times a^i.$$

Addendo	0 +	0 +	1 +	1 +
Addendo	0 =	1 =	0 =	1 =
Somma	0	1	1	0
Riporto	0	0	0	1

Figura A.8 Tabellina della somma binaria.

A.5 Aritmetica binaria

La Figura A.8 riporta la tabellina della somma per i numeri binari.

La somma di due addendi binari inizia dal bit meno significativo e procede sommando i bit che si trovano nelle posizioni corrispondenti. Se c'è un riporto, lo si somma nella colonna successiva a sinistra, proprio come nell'aritmetica decimale. Nell'aritmetica in complemento a uno, l'eventuale riporto generato dalla somma dei bit più significativi viene sommato al bit meno significativo. Questo procedimento si chiama *end-around carry* (cioè "riporto circolare"). Nell'aritmetica in complemento a due, l'eventuale riporto generato dalla somma dei bit più significativi viene semplicemente scartato. La Figura A.9 mostra due esempi di operazioni binarie.

Se gli addendi hanno segno opposto non si può verificare overflow. Se invece hanno lo stesso segno e il risultato ha segno opposto si è verificato un overflow e il risultato della somma è sbagliato. In entrambi i sistemi in complemento, si ha overflow se e soltanto se il resto in corrispondenza del bit di segno differisce dal riporto generato oltre il bit di segno. Molti calcolatori conservano questo bit in un apposito bit di overflow, ma il riporto del bit di segno non è desumibile dal risultato.

Decimale	In complemento a 1	In complemento a 2
10 +	00001010 +	00001010 +
(−3) =	11111100 =	11111101 =
+7	1 00000110	1 00000111
	riporto 1	scartato
	00000111	

Figura A.9 Somma in complemento a uno e in complemento a due.

Problemi

1. Si convertano in binario i numeri: 1984, 4000, 8192.
2. Qual è il numero decimale, ottale, esadecimale, rappresentato dalla stringa binaria 1001101001?
3. Quali delle seguenti stringhe

   ADA, BABBO, BARBA, CACCIA, DADA, DIDA, DECADE, EFFE,

   rappresentano un numero esadecimale?
4. Si esprima il numero decimale 100 in tutte le basi dalla 2 alla 9.
5. Quanti numeri diversi si possono rappresentare con k cifre in base b?
6. Molte persone riescono a contare solo fino a 10 solo usando le dita, ma gli informatici sanno fare di meglio. Se si guarda a ogni dito come a una cifra binaria, dove un dito teso indica 1 e un dito piegato verso il palmo indica 0, fino a che numero è possibile contare usando entrambe le mani? Con le mani e i piedi? Si usino ora le mani e i piedi per rappresentare i numeri in complemento a due, con l'alluce del piede sinistro impiegato come bit di segno. Qual è l'intervallo dei numeri rappresentabili? (*)
7. Si svolgano le operazioni seguenti su numeri di 8 bit in complemento a due.

00101101 +	11111111 +	00000000 +	11110111 +
01101111 =	11111111 =	11111111 =	11110111 =

   .....

   .....
8. Si svolga l'esercizio precedente nell'aritmetica in complemento a uno.
9. Si considerino le seguenti somme di numeri binari di 3 bit in complemento a due. Per ogni somma, si stabilisca se:
   a. il bit di segno del risultato è 1
   b. i tre bit meno significativi valgono 0
   a. si verifica un overflow.

000 +	000 +	111 +	100 +	100 +
001 =	111 =	110 =	111 =	100 =

   .....

   .....
10. I numeri decimali con segno lunghi n cifre possono essere rappresentati senza segno con $n + 1$ cifre. I numeri positivi hanno 0 come cifra più significativa, mentre i numeri negativi si ottengono a partire dai positivi sottraendo ciascuna loro cifra dalla cifra 9. Dunque la negazione di 014725 è 985274. Questa rappresentazione si chiama complemento a nove ed è analoga al complemento a uno dei numeri binari. Si esprimano i numeri seguenti come numeri di tre cifre in complemento a nove: 6, −2, 100, −14, −1, 0.

(*) Si declina ogni responsabilità sulle conseguenze di natura traumatico-ortopedica cui possono incorrere i lettori che svolgono meticolosamente questo esercizio (*N.d.Ren.*).

11. Si determini la regola della somma di numeri in complemento a nove e la si utilizzi sulle seguenti coppie di numeri:

0001 +	0001 +	9997 +	9241 +
9999 =	9998 =	9996 =	0802 =

.....

.....

12. Il complemento a dieci è analogo al complemento a due. Un numero negativo in complemento a dieci si ottiene sommando 1 al numero corrispondente in complemento a nove, ignorando l'eventuale resto. Qual è la regola della somma in complemento a dieci?

13. Si rediga la tabellina della moltiplicazione dei numeri in base 3.

14. Si moltiplichi in binario 0111 e 0011.

15. Si scriva un programma che accetti in ingresso un numero decimale con segno, sotto forma di stringa di caratteri ASCII, e stampi a schermo la sua rappresentazione in binario in complemento a due, in ottale e in esadecimale.

16. Si scriva un programma che prenda in ingresso due stringhe di 32 bit, sotto forma di caratteri ASCII. Il programma deve stampare la stringa di 32 caratteri ASCII corrispondente alla somma dei numeri, in complemento a 2, rappresentati dalle stringhe in ingresso.

Appendice B

Numeri in virgola mobile

In molti calcoli l'intervallo dei numeri utilizzati è molto esteso. Per esempio, un calcolo astronomico potrebbe riguardare la massa dell'elettrone, 9×10^{-28} grammi, e la massa del sole, 2×10^{33} grammi, con una differenza tra i valori superiore a 10^{60}. Questi due valori potrebbero essere rappresentati nella seguente forma

```
00000000000000000000000000000000,00000000000000000000000000000009
20000000000000000000000000000000,00000000000000000000000000000000
```

e tutti i calcoli potrebbero essere eseguiti mantenendo 34 cifre alla sinistra della virgola decimale e 28 posizioni alla sua destra. Così facendo i risultati sarebbero composti da 62 cifre significative. Su un calcolatore si potrebbe utilizzare l'aritmetica multiprecisione per fornire un numero sufficiente di cifre significative. Tuttavia la massa del sole non è conosciuta in modo accurato neanche fino alla quinta cifra, e tanto meno lo può essere fino alla sessantaduesima. In realtà poche misurazioni, di qualsiasi tipo esse siano, possono essere effettuate con (o richiedono una) precisione di 62 cifre significative. In teoria si potrebbero utilizzare tutte e 62 le cifre per i risultati intermedi per poi scartarne 50 o 60 prima di stampare i risultati finali, ma in pratica ciò non sarebbe altro che uno spreco di memoria e di tempo di CPU.

Quello di cui abbiamo bisogno è un sistema di rappresentazione dei numeri, nel quale l'intervallo di valori esprimibili sia indipendente dal numero di cifre significative. In questa appendice presenteremo un simile sistema, che si basa sulla notazione scientifica usata comunemente in fisica, chimica e ingegneria.

B.1 Principi dell'aritmetica in virgola mobile

Un modo per disaccoppiare l'intervallo dalla precisione consiste nell'esprimere i numeri nella notazione scientifica a noi familiare

$$n = f \times 10^e$$

dove f è chiamata **frazione**, o **mantissa**, ed e è un intero positivo o negativo chiamato **esponente**. La versione per il calcolatore di questa notazione è chiamata rappresentazione in **virgola mobile**. Alcuni esempi di numeri espressi in questa forma sono

$$3{,}14 = 0{,}314 \times 10^1 = 3{,}14 \times 10^0$$
$$0{,}000001 = 0{,}1 \times 10^{-5} = 1{,}0 \times 10^{-6}$$
$$1941 = 0{,}1941 \times 10^4 = 1{,}941 \times 10^3$$

L'intervallo è determinato dal numero di cifre che compongono l'esponente, mentre la precisione è determinata dal numero di cifre della frazione. Dato che un numero può essere rappresentato in più modi, in genere si sceglie un'unica forma come standard. Per analizzare le proprietà di cui gode questo metodo di rappresentazione dei numeri, consideriamo una rappresentazione, R, in cui la frazione vale zero oppure un valore con segno a tre cifre compreso nell'intervallo $0{,}1 \le |f| < 1$, mentre l'esponente è un valore con segno a due cifre. Questi numeri variano, in modulo, da $0{,}100 \times 10^{-99}$ a $0{,}999 \times 10^{+99}$ e, nonostante una variazione di quasi 199 ordini di grandezza, per essere memorizzati essi richiedendo solamente cinque cifre e due segni +.

I numeri in virgola mobile sono utilizzabili per modellare i numeri reali, anche se in modo impreciso. La Figura B.1 mostra uno schema, molto semplificato, della retta reale. Questa linea è divisa in sette regioni:

1. grandi numeri negativi minori di $-0{,}999 \times 10^{99}$
2. numeri negativi compresi tra $-0{,}999 \times 10^{99}$ e $-0{,}100 \times 10^{-99}$
3. piccoli numeri negativi con modulo minore di $0{,}100 \times 10^{-99}$
4. zero
5. piccoli numeri positivi con modulo minore di $0{,}100 \times 10^{-99}$
6. numeri positivi compresi tra $0{,}100 \times 10^{-99}$ e $0{,}999 \times 10^{99}$
7. grandi numeri positivi maggiori di $0{,}999 \times 10^{99}$.

Un'importante differenza tra i numeri reali e l'insieme di numeri rappresentabili usando tre cifre per la frazione e due per l'esponente è che in tal modo non si possono esprimere i numeri appartenenti alle regioni 1, 3, 5 e 7. Se il risultato di un'operazione aritmetica è un numero contenuto nelle regioni 1 oppure 7, per esempio $10^{60} \times 10^{60} = 10^{120}$, si verifica un **errore di overflow** e la risposta non sarà corretta. Il motivo è legato alla natura finita della rappresentazione dei numeri ed è inevitabile. È altrettanto impossibile esprimere un risultato che cade nella regione 3 oppure nella 5, una situazione chiamata **errore di underflow**. Questo errore è meno serio di quello di overflow, dato che spesso 0 è un'approssimazione soddisfacente per i numeri delle regioni 3 e 5. Infatti un saldo bancario di 10^{-102} euro non è molto migliore di uno di 0 euro.

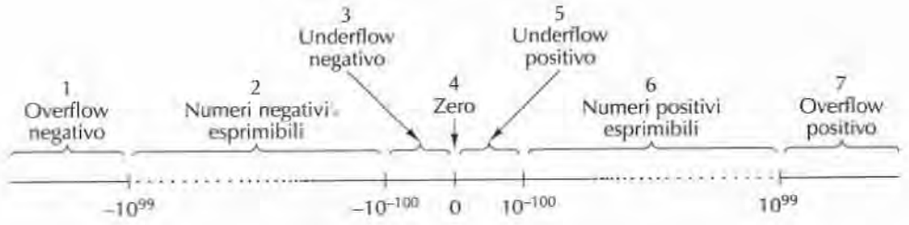


Figura B.1 La rappresentazione R ripartisce la retta reale in sette regioni.

Un'altra importante differenza tra i numeri in virgola mobile e i numeri reali è la loro densità. Tra due qualsiasi numeri reali, x e y, esiste un'altro numero reale, indipendentemente da quanto siano vicini x e y. Questa proprietà deriva dal fatto che per ogni coppia di numeri reali distinti, x e y, $z = (x + y)/2$ è un numero reale compreso fra x e y. I numeri reali formano un continuo.

Al contrario i numeri in virgola mobile sono un sistema discreto. Nel sistema a cinque cifre e due segni usato precedentemente è possibile esprimere esattamente 179.100 numeri positivi, 179.100 numeri negativi e il valore 0 (che può essere espresso in molti modi), per un totale di 358.201 numeri. Usando questa notazione è possibile quindi esprimere solamente 358.201 numeri degli infiniti numeri reali compresi tra -10^{100} e $+0{,}999 \times 10^{99}$. Essi sono rappresentati dai punti della Figura B.1. È molto probabile che il risultato di un calcolo, pur facendo parte delle regioni 2 o 6, sia uno degli altri numeri, quelli non rappresentabili. Per esempio $+0{,}100 \times 10^3$ diviso 3 non può essere espresso *esattamente* nel nostro sistema di rappresentazione. Se il risultato di un calcolo non può essere espresso nella rappresentazione utilizzata, la soluzione più ovvia consiste nell'utilizzare il più vicino numero rappresentabile. Questo processo è chiamato **arrotondamento**.

All'interno delle regioni 2 e 6 lo spazio tra numeri adiacenti non è costante. La distanza tra $+0{,}998 \times 10^{99}$ e $+0{,}999 \times 10^{99}$ è enormemente più grande della distanza tra $+0{,}998 \times 10^0$ e $+0{,}999 \times 10^0$. Tuttavia, se si esprime la distanza tra un numero e il suo successore come una percentuale di quel valore non vi è una sistematica variazione lungo le regioni 2 e 6. In altre parole l'**errore relativo** introdotto dall'arrotondamento è approssimativamente lo stesso sia per numeri piccoli sia per quelli grandi.

Cifre nella mantissa	Cifre nell'esponente	Limite inferiore	Limite superiore
3	1	10^{-12}	10^{9}
3	2	10^{-102}	10^{99}
3	3	10^{-1002}	10^{999}
3	4	10^{-10002}	10^{9999}
4	1	10^{-13}	10^{9}
4	2	10^{-103}	10^{99}
4	3	10^{-1003}	10^{999}
4	4	10^{-10003}	10^{9999}
5	1	10^{-14}	10^{9}
5	2	10^{-104}	10^{99}
5	3	10^{-1004}	10^{999}
5	4	10^{-10004}	10^{9999}
10	3	10^{-1009}	10^{999}
20	3	10^{-1019}	10^{999}

Figura B.2 Limiti inferiore e superiore approssimati dei numeri decimali (non normalizzati) esprimibili in virgola mobile.

Anche se queste considerazioni sono basate su una rappresentazione con tre cifre per la mantissa e due cifre per l'esponente, le conclusioni tratte valgono anche per altri sistemi di rappresentazione. Modificando il numero di cifre della frazione o dell'esponente si spostano semplicemente i limiti delle regioni 2 e 6 e si modifica il numero di punti esprimibili al loro interno. Aumentando il numero di cifre della frazione si aumenta la densità dei punti e quindi si migliora il grado di accuratezza dell'approssimazione. Aumentando le cifre dell'esponente si aumenta la dimensione delle regioni 2 e 6 riducendo le regioni 1, 3, 5 e 7. La Figura B.2 mostra, in modo approssimato, i limiti della regione 6 per numeri decimali in virgola mobile che utilizzano dimensioni diverse per la frazione e l'esponente.

Nei calcolatori si utilizza una variante di questa rappresentazione. Per ragioni di efficienza l'elevamento a potenza viene fatto utilizzando come base 2, 4, 8 o 16, e non 10: la frazione consiste quindi in una stringa di cifre binarie, in base 4, ottali o esadecimali. Se la cifra più a sinistra è zero è possibile traslare di una posizione a sinistra tutte le cifre e diminuire di 1 l'esponente senza variare il valore del numero (a parte l'underflow). Una frazione in cui la cifra più a sinistra è diversa da zero è detta **normalizzata**.

Generalmente i numeri normalizzati sono preferibili a quelli non normalizzati, dato che dei primi esiste un'unica forma, mentre dei secondi ce ne sono molteplici. La Figura B.3 mostra un numero in virgola mobile normalizzato con due basi diverse per l'elevamento a potenza. Negli esempi è stata utilizzata una rappresentazione in cui la frazione usa 16 bit (compreso il bit del segno) e l'esponente usa 7 bit in notazione in eccesso 64.

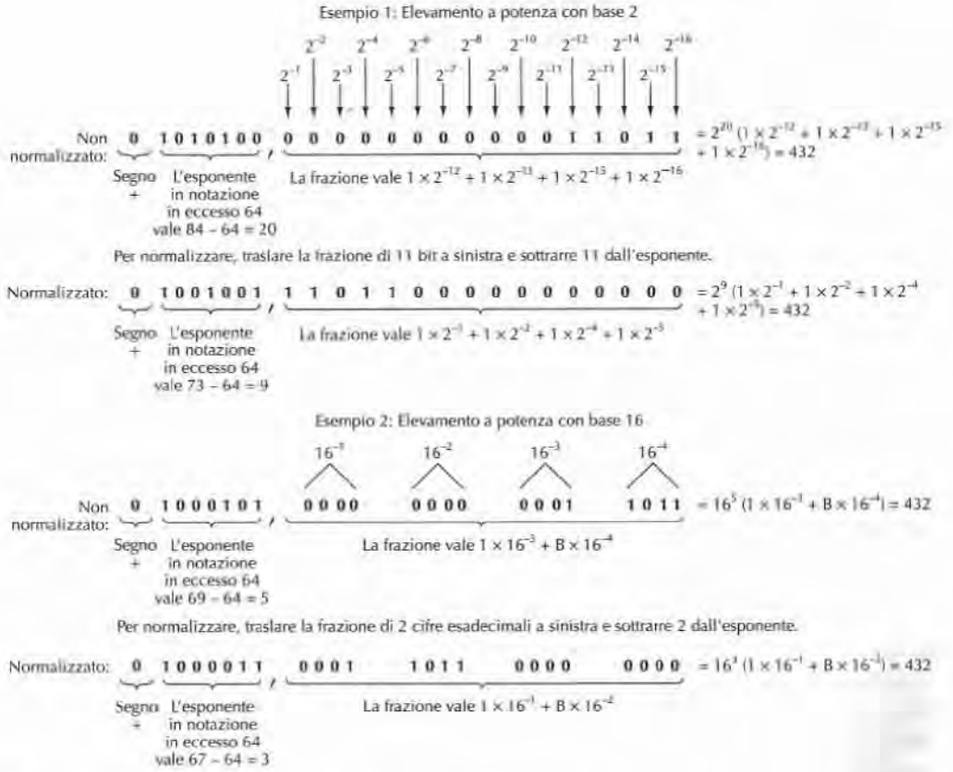


Figura B.3 Esempi di numeri in virgola mobile normalizzati.

B.2 Standard in virgola mobile IEEE 754

Fino al 1980 ogni produttore di calcolatori aveva il proprio formato di aritmetica in virgola mobile che, non c'è bisogno di dirlo, erano tutti diversi l'uno dall'altro. Ancor peggio, alcune di queste rappresentazioni non eseguivano correttamente i calcoli aritmetici; esistono infatti alcune finezze dell'aritmetica in virgola mobile che non appaiono così ovvie a tutti i progettisti hardware.

Alla fine degli anni '70, per porre rimedio alla situazione, venne costituito un comitato per standardizzare l'aritmetica in virgola mobile. L'obiettivo non era solo quello di permettere a calcolatori diversi di scambiarsi dati in virgola mobile, ma anche quello di fornire ai progettisti dell'hardware un modello di cui si conosceva la correttezza. Il risultato del lavoro fu lo standard IEEE 754 (IEEE, 1985). La maggior parte dei calcolatori attuali (compresi i modelli Intel, SPARC e JVM studiati in questo testo) ha istruzioni in virgola mobile

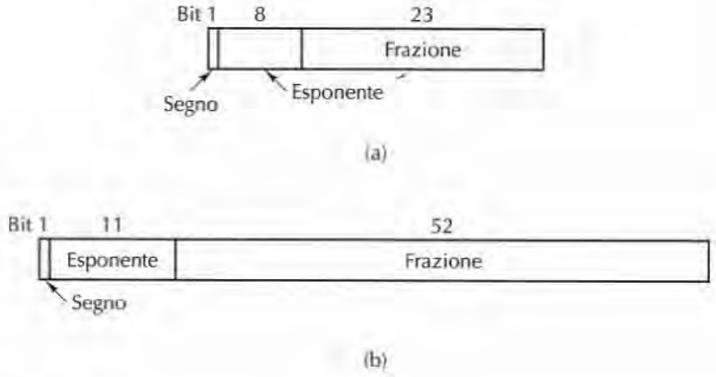


Figura B.4 Formato IEEE in virgola mobile. (a) Precisione singola. (b) Precisione doppia.

conformi a questo standard. Diversamente da molti altri standard che tendono a essere soltanto dei compromessi annacquati che non accontentano nessuno, questo standard non è male; il motivo principale è che fu il prodotto del lavoro di una sola persona: il professor William Kahan, un matematico di Berkely.

Lo standard definisce tre formati: precisione singola (32 bit), precisione doppia (64 bit) e precisione estesa (80 bit). Quest'ultimo è stato pensato per ridurre gli errori di arrotondamento ed è usato principalmente nelle unità aritmetiche in virgola mobile; per questo motivo non lo tratteremo ulteriormente. I primi due (mostrati nella Figura B.4) usano la base 2 per le frazioni e la notazione in eccesso per gli esponenti.

Entrambi i formati iniziano con un bit che rappresenta il segno dell'intero numero, 0 se è positvo e 1 se è negativo. Di seguito vi è l'esponente, che utilizza la notazione in eccesso 127 per la precisione singola e la notazione in eccesso 1023 per la precisione doppia. Gli esponenti minimo (0) e massimo (255 e 2047) non sono utilizzati per i numeri normalizzati; essi hanno un utilizzo speciale che descriveremo in seguito. Infine troviamo le frazioni, rispettivamente di 23 e 52 bit.

Una frazione normalizzata inizia con una virgola, seguita da un bit 1 e poi dal resto della frazione. Seguendo una pratica iniziata con il PDP-11 gli autori dello standard hanno deciso che il bit 1 che dovrebbe precedere la virgola non doveva essere memorizzato, dato che se ne poteva implicitamente assumere la presenza. Lo standard definisce quindi la frazione in modo leggermente diverso dal solito; consiste in un bit 1 sottointeso, in una virgola binaria sottointesa e poi in 23 (o 52) bit arbitrari. Se tutti e 23 (o 52) i bit della frazione valgono 0, il valore numerico della frazione è 1,0. Se tutti valgono 1, il valore numerico della frazione è leggermente inferiore a 2,0. Per evitare confusione con la forma convenzionale della frazione, la combinazione del valore 1 sottointeso, della virgola binaria sottointesa e dei 23, o 52, bit espliciti è chiamata **significando**, invece di frazione o mantissa. Tutti i numeri normalizzati hanno un significando, s, compreso nell'intervallo [1,2].

Le caratteristiche numeriche dello standard sono elencate nella Figura B.5. Come esempio consideriamo i numeri 0,5, 1 e 1,5 nel formato normalizzato in precisione singola. Le loro rappresentazioni in esadecimale sono rispettivamente 3F000000, 3F800000 e 3FC00000.

Caratteristica	Precisione singola	Precisione doppia
Bit nel segno	1	1
Bit nell'esponente	8	11
Bit nella frazione	23	52
Bit totali	32	64
Notazione dell'esponente	in eccesso 127	in eccesso 1023
Intervallo dell'esponente	da –126 a +127	da –1022 a +1023
Numero normalizzato più piccolo	2^{-126}	2^{-1022}
Numero normalizzato più grande	circa 2^{128}	circa 2^{1024}
Intervallo decimale	circa da 10^{-38} a 10^{38}	circa da 10^{-308} a 10^{308}
Numero denormalizzato più grande	circa 10^{-45}	circa 10^{-324}

Figura B.5 Caratteristiche dell'aritmetica IEEE in virgola mobile.

Uno dei tipici problemi dell'aritmetica in virgola mobile è il modo di gestire l'underflow, l'overflow e i numeri non inizializzati. Lo standard IEEE tratta questi problemi in modo esplicito, traendo parte del proprio approccio da quello utilizzato nel calcolatore CDC 6600. Oltre ai numeri normalizzati lo standard definisce quattro altri tipi numerici, descritti in seguito e mostrati nella Figura B.6.

Quando il risultato di un calcolo ha un modulo minore del più piccolo numero in virgola mobile normalizzato sorge un problema, dato che questo sistema non può rappresentarlo.

Prima della definizione dello standard IEEE le scelte più comuni erano: impostare il risultato a zero e continuare oppure generare un'eccezione di underflow. Dato che nessuna di queste soluzioni era veramente soddisfacente, IEEE ha inventato i **numeri denormalizzati**. Questi numeri hanno come esponente 0 e come frazione i successivi 23, o 52, bit. Il bit 1 implicito alla sinistra della virgola binaria diventa ora 0. È possibile distinguere i numeri normalizzati da quelli denormalizzati, dato che gli ultimi non possono avere un valore 0 come esponente.

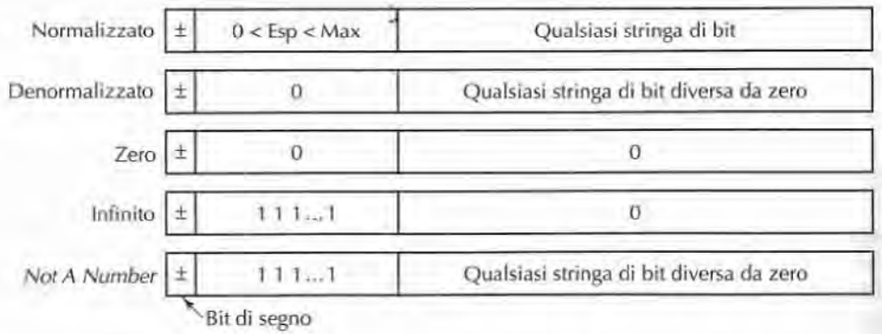


Figura B.6 Tipi di numeri IEEE.

Il più piccolo numero normalizzato in precisione singola ha 1 come esponente e 0 come frazione, e rappresenta il numero $1{,}0 \times 2^{-126}$. Il più grande numero denormalizzato ha 0 come esponente e tutti i bit della frazione e impostati a 1, e rappresenta circa $0{,}9999999 \times 2^{-126}$, che è praticamente uguale al valore precedente. Occorre tuttavia notare che questo numero ha solo 23 bit significativi, contro i 24 dei numeri normalizzati.

Al diminuire del risultato dei calcoli l'esponente rimane fisso a 0, mentre si azzerano progressivamente i primi bit della frazione, riducendo in questo modo sia il valore rappresentato sia il numero di bit significativi della frazione. Il più piccolo numero denormalizzato diverso da zero consiste in un valore 1 nel bit più a destra, con tutti gli altri bit pari a 0. Dato che l'esponente rappresenta 2^{-126} e la frazione rappresenta 2^{-23}, il valore è 2^{-149}. Questo schema rende meno grave il problema dell'underflow, dato che quando il risultato non può essere espresso come un risultato normalizzato non si salta improvvisamente al valore 0, ma il numero di cifre significative diminuisce progressivamente.

In questo schema lo zero ha due rappresentazioni, una positiva e una negativa, determinate dal bit del segno. Entrambe hanno 0 come esponente e come frazione. Anche in questo caso il bit a sinistra della virgola binaria è implicitamente 0 invece di 1.

L'overflow non può essere gestito in modo graduale come l'underflow, dato che non ci sono altre combinazioni di bit. Esiste però una rappresentazione per l'infinito che consiste in un esponente in cui tutti i bit valgono 1 (non permesso per i numeri normalizzati) e di una frazione che vale 0. Questo numero segue le regole matematiche dell'infinito. Per esempio la somma tra infinito e qualsiasi altro valore dà come risultato infinito e qualsiasi numero finito diviso infinito dà come risultato zero. Analogamente ogni numero finito diviso zero dà come risultato infinito.

E se si divide infinito per infinito? Il risultato non è definito. Per gestire questo caso è previsto un ulteriore formato, chiamato **NaN (Not a Number)**. Anch'esso può essere usato come operando con risultati prevedibili.

Problemi

1. Si convertano i seguenti numeri nel formato IEEE in precisione singola. Si scriva il risultato in esadecimale.
   a. 9
   b. 5/32
   c. –5/32
   d. 6,125
2. Si convertano da esadecimali a decimali i seguenti numeri rappresentati nel formato IEEE in virgola mobile in precisione singola:
   a. 42E48000H
   b. 3F880000H
   c. 00800000H
   d. C7F00000H
3. Sul calcolatore 370 il formato dei numeri in virgola mobile in precisione singola è composto da un esponente di 7 bit in notazione in eccesso 64 e da una frazione contenente 24 bit più un bit per il segno; la virgola si trova all'estremità sinistra della frazione. La base dell'elevamento a potenza è 16. L'ordine dei campi è: bit del segno, esponente, frazione. Si esprima, in esadecimale, il numero 7/64 come un numero normalizzato in questo sistema.
4. I seguenti numeri binari in virgola mobile consistono in un bit del segno, un esponente di una base 2 espresso in notazione in eccesso 64 e una frazione a 16 bit. Li si normalizzi.
   a. 0 1000000 0001010100000001
   b. 0 0111111 0000001111111111
   c. 0 1000011 1000000000000000
5. Per sommare due numeri in virgola mobile occorre modificare gli esponenti (traslando i bit della frazione) affinché abbiano lo stesso valore. In seguito è possibile sommare le frazioni e, se necessario, normalizzare il risultato. Si sommino i numeri IEEE in precisione singola 3EE00000H e 3D800000H e si esprima in esadecimale il risultato normalizzato.
6. La Società di Calcolatori BraccioCorto ha deciso di produrre una macchina con numeri in virgola mobile a 16 bit. Il formato in virgola mobile del Modello 0,001 ha un bit per il segno, 7 bit per l'esponente in notazione in eccesso 64 e 8 bit per la frazione. Il formato del Modello 0,002 ha un bit per il segno, 5 bit per l'esponente in notazione in eccesso 16 e 10 bit per la frazione. Entrambi usano la base 2 per l'elevamento a potenza. Quali sono i numeri più piccoli e più grandi rappresentabili sui due modelli? Quante cifre decimali di precisione ha, all'incirca, ciascuno di loro? Comprereste uno di questi modelli?
7. Esiste una situazione nella quale un'operazione su due numeri in virgola mobile possa causare una riduzione drastica del numero di bit significativi nel risultato. Qual è?
8. Alcuni coprocessori in virgola mobile hanno un'istruzione predefinita per la radice quadrata. Un algoritmo possibile è di tipo iterativo (per esempio, Newton-Raphson). Gli algoritmi iterativi hanno bisogno di un'approssimazione iniziale del risultato che poi migliorano costantemente. Come si può ottenere una veloce approssimazione della radice quadrata di un numero in virgola mobile?

9. Si scriva una procedura per sommare due numeri IEEE in virgola mobile in precisione singola. Ciascun numero è rappresentato da un array di 32 elementi booleani.

10. Si scriva una procedura per sommare due numeri in virgola mobile in precisione singola che usano la base 16 per l'esponente e la base 2 per la frazione, ma non hanno un bit 1 sottointeso alla sinistra della virgola. Un numero normalizzato ha, nei quattro bit più a sinistra della frazione, 0001, 0010, ..., 1111, ma non 0000. Un numero viene normalizzato traslando a sinistra di 4 bit la frazione e sottraendo 1 all'esponente.

Appendice C

Programmazione in linguaggio assemblativo

Evert Wattel
Vrije Universiteit
Amsterdam, Olanda

Ogni calcolatore ha un livello **ISA** (*Instruction Set Architecture*, "architettura dell'insieme d'istruzioni") che riguarda registri, istruzioni e altre caratteristiche visibili ai programmatori di linguaggi a basso livello. Spesso ci si riferisce al livello ISA con il termine di **linguaggio macchina**, benché non sia del tutto corretto. Un programma a questo livello di astrazione è una lunga lista di stringhe binarie, una per istruzione, che specificano le istruzioni da eseguire e i loro operandi. Programmare in binario è molto astruso, perciò tutti i calcolatori dispongono di un **linguaggio assemblativo** (*assembly language*), cioè di una rappresentazione simbolica dell'architettura dell'insieme d'istruzioni che usa nomi quali `ADD`, `SUB` e `MUL` al posto delle stringhe binarie. Questa appendice tratta la programmazione in linguaggio assemblativo dell'Intel 8088 (il processore utilizzato nei primi PC di IBM e che costituì la base per lo sviluppo dei Pentium moderni) e l'uso di alcuni strumenti disponibili nel materiale di supporto allegato al testo.

Il nostro scopo non è formare nuovi *programmatori assembler* (questa è la locuzione gergale), ma assistere il lettore nell'esplorazione dell'architettura dei calcolatori mettendolo in grado di "toccare con mano" la materia. A tal fine, abbiamo scelto come esempio della trattazione una macchina molto semplice. Anche se è oramai impossibile procurarsi questi processori, tutti i Pentium possono eseguire i programmi scritti per l'8088 e quindi le lezioni che impartiremo valgono ancora per le macchine moderne. Inoltre, molte delle istruzioni principali del Pentium sono uguali a quelle dell'8088, con la sola differenza che usano registri a 32 bit invece che a 16. Dunque questa appendice può essere considerata una graduale introduzione alla programmazione nel linguaggio assemblativo del Pentium.

La programmazione di una macchina in linguaggio assemblativo prevede che il programmatore abbia una conoscenza approfondita dell'architettura dell'insieme d'istruzioni della macchina. Di conseguenza, i primi quattro paragrafi sono dedicati all'architettura dell'8088, alla sua organizzazione di memoria, alle modalità d'indirizzamento e alle istruzioni. Il Paragrafo C.5 descrive l'assemblatore impiegato in questa appendice, reperibile nel CD-ROM allegato al testo, e noi ci atterremo alla notazione usata da questo assemblatore. I lettori abituati alla notazione del linguaggio assemblativo dell'8088 dovrebbero tener presente che altri assemblatori usano notazioni diverse. Il Paragrafo C.6 illustra lo strumento interprete/tracer/debugger che assiste il programmatore nel debugging dei programmi e che si trova ugualmente nel materiale di supporto al testo. Il Paragrafo C.7 concerne l'installazione degli strumenti e la guida al loro utilizzo. Il Paragrafo C.8 contiene alcuni programmi, esempi, eser-

cizi e soluzioni. Il Paragrafo C.9 è una digressione su alcune questioni implementative, sui bachi e sulle limitazioni del materiale fornito.

C.1 Panoramica

Cominciamo la nostra visita guidata alla programmazione in linguaggio assemblativo spendendo qualche parola sul linguaggio stesso e fornendo poi qualche esempio.

C.1.1 Linguaggio assemblativo

Ogni assemblatore utilizza **nomi mnemonici**, cioè delle abbreviazioni quali `ADD`, `SUB` e `MUL` per denotare alcune entità e renderle più facili da ricordare: in questo caso addizione, sottrazione e moltiplicazione. Inoltre, gli assemblatori usano **nomi simbolici** per variabili e costanti, ed **etichette** per le istruzioni e per gli indirizzi di memoria. Molti assemblatori supportano poi alcune **direttive** che non vengono tradotte in istruzioni ISA, ma rappresentano comandi impartiti all'assemblatore per guidarne l'attività.

L'**assemblatore** (*assembler*) è un programma che riceve in ingresso un file contenente un programma in linguaggio assemblativo, e genera un file contenente un **programma binario** adatto all'esecuzione vera e propria, cioè pronto per essere eseguito dall'hardware. Tuttavia, i principianti della programmazione in linguaggio assemblativo commettono spesso degli errori che causano l'interruzione improvvisa dell'esecuzione, senza alcuna indicazione circa il problema che si è verificato.

Per semplificare la vita dei principianti, è possibile far eseguire il programma binario non all'hardware vero e proprio, ma a un simulatore che esegue un'istruzione per volta e visualizza un resoconto dettagliato della sua attività. Così facendo si semplifica notevolmente l'attività di debugging. I programmi eseguiti dai simulatori sono ovviamente molto lenti, ma ciò non costituisce un problema se l'obiettivo è quello di apprendere la programmazione in linguaggio assemblativo. Questa appendice si basa su vari strumenti di sviluppo tra cui un simulatore chiamato **interprete,** o **tracer**, poiché interpreta e traccia l'esecuzione del programma binario un'istruzione alla volta. I termini "simulatore", "interprete" e "tracer" saranno usati in modo del tutto interscambiabile nel corso della trattazione. In genere parleremo di "interprete" quando ci riferiremo alla sola esecuzione di un programma, mentre parleremo di "tracer" quando ci riferiremo allo strumento di debugging, ma si tratta sempre dello stesso programma.

C.1.2 Breve programma in linguaggio assemblativo

Concretizziamo queste idee astratte considerando il programma e il tracer della Figura C.1. La figura illustra una schermata del tracer e un breve programma nel linguaggio assemblativo dell'8088. I numeri che si trovano dopo i punti esclamativi sono i numeri delle righe di codice e fanno riferimento a parti del programma. Il CD allegato al testo contiene una copia di questo programma nel file *HlloWrld.s* all'interno della directory *examples*. Questo programma ha estensione *.s*, come tutti i programmi assemblativi trattati in questa appendice, a indicare il fatto che si tratta del sorgente di un programma in linguaggio assemblativo. La schermata del tracer, mostrata nella Figura C.1(b), contiene sette finestre, ciascuna delle quali riporta alcune informazioni sullo stato del programma in esecuzione.

Esaminiamo brevemente la Figura C.1(b). La finestra in alto a sinistra mostra il contenuto del processore, ovvero il valore corrente dei registri di segmento `CS`, `DS`, `SS` ed `ES`, dei registri aritmetici `AH`, `AL`, `AX` e di altri.

La finestra in alto al centro rappresenta lo stack, l'area di memoria per le variabili temporanee.

La finestra in alto a destra contiene un frammento del programma in linguaggio assemblativo, e la freccia indica l'istruzione man mano in esecuzione. La comodità del tracer sta nel fatto che basta premere il tasto Invio per eseguire un'istruzione e aggiornare simultaneamente tutte le finestre, perciò è possibile eseguire il programma un'istruzione per volta.

Al di sotto della finestra di sinistra ce n'è una contenente lo stack delle chiamate di subroutine (in questo caso è vuota). Ancora più in basso vediamo i comandi del tracer. A destra di queste due finestre si trova la finestra per l'input/output e per i messaggi d'errore.

La finestra al fondo della figura mostra una porzione della memoria. Più avanti descriveremo queste finestre in maggior dettaglio, ma dovrebbe essere già chiara l'idea fondamentale: il tracer mostra il sorgente del programma, i registri di macchina e varie informazioni sullo stato del programma in esecuzione. Le informazioni vengono aggiornate dopo l'esecuzione di ogni istruzione, permettendo all'utente di osservare il funzionamento del programma in ogni dettaglio.

```
   _EXIT = 1                  !1
   _WRITE = 4                 !2
   _STDOUT = 1                !3
.SECT .TEXT                   !4
start:                        !5
   MOV  CX,de-hw              !6
   PUSH CX                    !7
   PUSH hw                    !8
   PUSH _STDOUT               !9
   PUSH _WRITE                !10
   SYS                        !11
   ADD  SP, 8                 !12
   SUB  CX,AX                 !13
   PUSH CX                    !14
   PUSH _EXIT                 !15
   SYS                        !16
.SECT .DATA                   !17
hw:                           !18
.ASCII "Hello World\n"        !19
de: .BYTE 0                   !20
```

(a)

```
CS: 00    DS=SS=ES: 002
AH:00  AL:0c   AX:      12
BH:00  BL:00   BX:       0
CH:00  CL:0c   CX:      12
DH:00  DL:00   DX:       0
SP:  7fd8  SF  O D S Z C    =>0004
BP:  0000  CC  - > p - -      0001   =>
SI:  0000   IP:000c:PC        0000
DI:  0000   start + 7         000c

                                      MOV  CX,de-hw   !6
                                      PUSH CX         !7
                                      PUSH HW         !8
                                      PUSH _STDOUT    !9
                                      PUSH _WRITE     !10
                                      SYS             !11
                                      ADD  SP,8       !12
                                      SUB  CX,AX      !13
                                      PUSH CX         !14

                          E
                          l

hw
■                         > Hello  World\n

hw + 0 = 0000:  48  65  6c  6c  6f  20  57  6f  Hello  World  25928
```

(b)

Figura C.1 (a) Programma in linguaggio assemblativo. (b) Schermata del tracer corrispondente alla sua esecuzione.

C.2 Processore 8088

Tutti i processori sono provvisti di uno stato interno contenente alcune informazioni cruciali per il loro funzionamento. A tal fine, il processore dispone di un insieme di **registri** dove memorizzare ed elaborare queste informazioni. Probabilmente il registro più importante è il PC (**program counter**), contenente la locazione di memoria della successiva istruzione da eseguire, ovvero il suo **indirizzo**. A volte, questo registro è chiamato IP (**instruction pointer**). Le istruzioni da eseguire si trovano in una parte della memoria detta **segmento di codice**. La memoria principale dell'8088 può superare di poco il megabyte, ma il segmento di codice corrente è di soli 64 KB. Il registro CS della Figura C.1 indica l'inizio del segmento di codice all'interno della memoria (da 1 MB). È possibile attivare un nuovo segmento di codice semplicemente cambiando il contenuto del registro CS. Allo stesso modo, esiste anche un segmento dati di 64 KB contenente i dati del programma. Il registro DS punta alla prima locazione del segmento dati e anch'esso può essere modificato per accedere a dati esterni al segmento dati corrente. I registri CS e DS sono necessari perché l'8088 ha registri di 16 bit che non possono memorizzare direttamente i 20 bit di un indirizzo di una memoria di 1 MB. È questa la ragione per cui furono introdotti i registri di segmento.

Gli altri registri contengono i dati e i puntatori ai dati che si trovano nella memoria principale. I programmi in linguaggio assemblativo possono accedere direttamente a questi registri. Oltre ai registri, il processore contiene tutta la circuiteria per l'esecuzione delle istruzioni, ma questa è accessibile al programmatore solo attraverso la dichiarazione d'istruzioni.

C.2.1 Ciclo del processore

L'attività dell'8088 (e degli altri calcolatori) consiste nell'esecuzione d'istruzioni in rapida successione. L'esecuzione di una singola istruzione può essere scomposta nelle fasi seguenti:

1. fetch dell'istruzione dalla memoria (in particolare dal segmento di codice) tramite il PC
2. incremento del program counter
3. decodifica dell'istruzione prelevata
4. fetch dei dati necessari dalla memoria e/o dai registri del processore
5. svolgimento dell'istruzione
6. memorizzazione dei risultati dell'istruzione in memoria e/o nei registri
7. ritorno alla fase 1 per l'avvio dell'istruzione successiva.

L'esecuzione di un'istruzione è simile a quella di un programma in miniatura. Infatti, alcune macchine usano davvero un programma molto piccolo, il **microprogramma**, per l'esecuzione delle loro istruzioni. I microprogrammi sono descritti ampiamente nel Capitolo 4.

Dal punto di vista del programmatore in linguaggio assemblativo, l'8088 ha 14 registri che costituiscono in un certo senso lo spazio di lavoro a disposizione delle istruzioni, anche se i dati ivi memorizzati vengono modificati molto spesso e non sono quindi molto adatti a contenere risultati definitivi. La Figura C.2 dà una visione d'insieme dei 14 registri molto

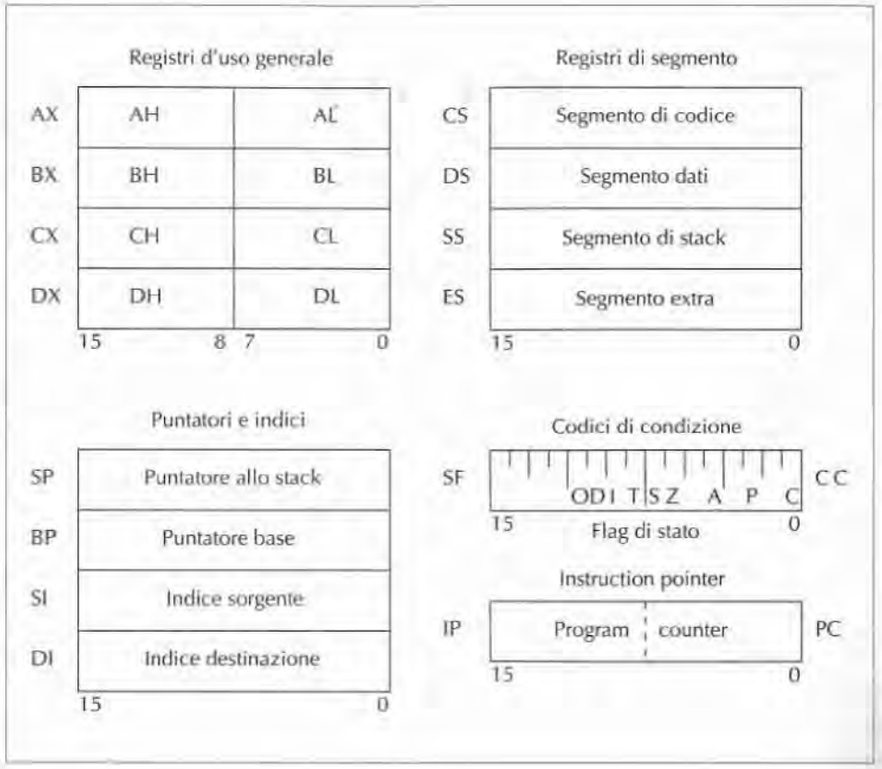


Figura C.2 Registri dell'8088.

simile alla finestra dei registri del tracer mostrata nella Figura C.1. Infatti le due figure rappresentano la stessa informazione.

I registri dell'8088 sono ampi 16 bit e svolgono tutti funzioni diverse. Alcuni di loro però condividono alcune caratteristiche, perciò è possibile individuare i gruppi della Figura C.2. Esaminiamo ciascun gruppo separatamente.

C.2.2 Registri d'uso generale

I registri del primo gruppo, AX, BX, CX e DX, sono **d'uso generale**. Il primo di questi, AX, si chiama **registro accumulatore** (o semplicemente **accumulatore**) ed è usato per accogliere il risultato dell'elaborazione e funge da destinazione di un gran numero d'istruzioni. Anche se molte istruzioni possono specificare un registro qualsiasi per ospitare il loro risultato, altre definiscono implicitamente AX come destinazione del risultato dell'operazione, come fa per esempio la moltiplicazione.

Il secondo registro del gruppo è BX, il **registro base**. BX può essere usato al posto di AX per molte finalità, ma ha anche una funzionalità che AX non ha: è possibile memoriz-

zare in BX un indirizzo di memoria e poi eseguire un'istruzione il cui operando proviene dall'indirizzo di memoria contenuto in BX. Detto altrimenti, BX può contenere un puntatore alla memoria, ma AX no. Vediamo questa distinzione mediante due istruzioni. Per prima cosa scriviamo

```
MOV AX,BX
```

che copia in AX il contenuto di BX. Poi scriviamo

```
MOV AX,(BX)
```

che copia in AX il contenuto della parola di memoria che si trova all'indirizzo specificato da BX. Nel primo esempio BX contiene l'operando sorgente; nel secondo caso invece punta all'operando sorgente. In entrambi gli esempi l'istruzione MOV contiene un operando sorgente e uno destinazione, e la destinazione è indicata prima della sorgente.

Il registro CX è il **registro contatore**. È usato per diversi compiti, e in particolare come contatore dei cicli. Viene decrementato automaticamente dall'istruzione LOOP e i cicli terminano quando CX raggiunge il valore zero.

Il quarto registro d'uso generale è DX, il **registro dati**, usato congiuntamente ad AX per contenere le istruzioni lunghe due parole (32 bit). In tal caso DX contiene i 16 bit più significativi e AX gli altri. Generalmente, gli interi di 32 bit si dicono **long**, mentre il termine **double** è riservato ai valori in virgola mobile di 64 bit, benché qualcuno usi il termine "double" per riferirsi agli interi di 32 bit. In queste pagine non ci sarà alcuna ambiguità per il semplice fatto che non trattiamo i numeri in virgola mobile.

Tutti i registri d'uso generale possono essere visti come registri di 16 bit o come coppie di registri di 8 bit. In tal modo l'8088 dispone esattamente di 8 registri di 8 bit utilizzabili per le istruzioni che trattano byte o caratteri. Nessun altro registro può essere suddiviso in due da 8 bit. È importante capire che AL e AH sono semplicemente i nomi delle due metà di AX. Quando si carica un nuovo valore in AX, AL e AH sono entrambi modificati per contenere rispettivamente la metà inferiore e la metà superiore del numero di 16 bit inserito in AX. Per comprendere la relazione tra AX, AH e AL si consideri l'istruzione

```
MOV AX,258
```

che scrive il valore decimale 258 nel registro AX. Dopo l'esecuzione dell'istruzione, il registro di un byte AH contiene il valore 1 e il registro di un byte AL contiene il numero 2. Infatti, i 16 bit della rappresentazione binaria di 258 sono

00000001 00000010.

Se l'istruzione è seguita dall'istruzione di somma tra byte

```
ADDB AH,AL
```

allora il registro AH viene incrementato con il valore di AL (2) e quindi ora vale 3. L'effetto di questa azione è che ora il registro AX contiene il valore 770, equivalente a

00000011 00000010 (binario), ovvero 03 02 (esadecimale).

Gli otto registri da un byte sono quasi completamente interscambiabili, con l'eccezione che AL contiene sempre uno degli operandi dell'istruzione MULB ed è la destinazione implicita di questa operazione, insieme ad AH. Anche DIVB usa la coppia AH:AL per contenere il dividendo. Il byte meno significativo del registro contatore, CL, può essere usato per contenere il numero di cicli delle istruzioni di rotazione e scorrimento.

Il secondo esempio del Paragrafo C.8 mostra le proprietà dei registri d'uso generale attraverso la presentazione del programma *GenReg.s*.

C.2.3 Registri puntatore

Il secondo gruppo di registri contiene i **registri puntatore** e i **registri indice**. Tra questi, il più importante è il **puntatore allo stack** denotato con SP. Uno stack è un segmento di memoria contenente informazioni sul contesto del programma in esecuzione. All'atto di una chiamata di procedura, una parte dello stack viene riservata alla memorizzazione delle sue variabili locali, dell'indirizzo dell'istruzione da eseguire dopo il suo completamento, e di altre informazioni di controllo. La parte di stack che riguarda una procedura si chiama il suo **record d'attivazione** (*stack frame*). Quando una procedura ne chiama un'altra, viene allocato un altro record d'attivazione di seguito a quelli già presenti. Pur non essendo una regola obbligatoria, in genere gli stack crescono verso il basso, dagli indirizzi maggiori verso i minori. Ciò nondimeno, l'ultimo indirizzo occupato dallo stack si chiama sempre cima dello stack.

Oltre a ospitare le variabili locali, gli stack possono servire anche per memorizzare i risultati temporanei. L'8088 dispone dell'istruzione PUSH per impilare una parola di 16 bit in cima allo stack. L'istruzione comincia con il decrementare SP di 2, per poi memorizzare il suo operando al nuovo indirizzo puntato da SP. Analogamente, l'istruzione POP rimuove una parola di 16 bit dalla cima dello stack prelevando il valore che si trova in cima e incrementando SP di 2. Il registro SP punta sempre alla cima dello stack e viene modificato dalle istruzioni PUSH, POP e CALL; in particolare viene decrementato da PUSH e CALL, e incrementato da POP.

Il registro successivo è BP, il **puntatore base**, che in genere punta a una locazione dello stack. A differenza di SP, che punta sempre alla cima dello stack, BP può puntare a una sua locazione qualsiasi. Nella pratica viene usato comunemente per puntare all'inizio del record d'attivazione della procedura corrente, per raggiungere le sue variabili locali. Dunque BP punta spesso alla base del record d'attivazione corrente (al suo indirizzo più grande) e SP alla sua cima (al suo indirizzo più piccolo), perciò questi due registri delimitano il record d'attivazione.

Appartengono allo stesso gruppo i due registri indice: SI, **indice sorgente** e DI, **indice destinazione**. Questi registri sono usati spesso in combinazione con BP per riferirsi ai dati dello stack, o con BX per localizzare i dati in memoria. Per una loro più ampia trattazione rimandiamo al paragrafo sulle modalità d'indirizzamento.

Uno dei registri più importanti, tanto da costituire un gruppo a sé, è l'**instruction pointer**, l'equivalente Intel del program counter (PC). Questo registro non viene usato direttamente dalle istruzioni, bensì contiene un indirizzo di memoria appartenente al segmento di codice del programma. Il ciclo del processore comincia con il fetch dell'istruzione puntata da PC, dopodiché questo viene incrementato prima del completamento dell'istruzione in esecuzione. Così facendo questo registro punta sempre all'istruzione che segue quella corrente.

Il **registro di flag**, o **dei codici di condizione**, è in pratica un insieme di registri da 1 bit. Alcuni di questi bit sono impostati dalle istruzioni aritmetiche e rappresentano il loro risultato, nei modi seguenti:

Z – il risultato è zero
S – il risultato è negativo (bit di segno)
V – il risultato ha causato un overflow
C – il risultato ha generato un riporto
A – riporto ausiliario (oltre il bit 3)
P – parità del risultato.

Gli altri bit del registro controllano certi aspetti dell'attività del processore. Il bit *I* attiva gli interrupt. Il bit *T* abilità la modalità di tracing, usata per il debugging, e il bit *D* controlla la direzione delle operazioni su stringhe. Non tutti i 16 bit del registro sono utilizzati; quelli inutilizzati sono cablati al valore zero.

Segue poi il gruppo dei quattro **registri di segmento**. Ricordiamo che i segmenti dello stack, dei dati e delle istruzioni risiedono tutti in memoria principale, ma spesso in posti diversi. I registri di segmento servono alla gestione di queste diverse porzioni della memoria che prendono il nome di **segmenti**. Questi registri comprendono CS per il segmento di codice, DS per il segmento dati, SS per il segmento di stack ed ES per il segmento extra. Il loro valore resta costante per la maggior parte del tempo. In quasi tutte le implementazioni, il segmento dati e il segmento di stack usano la stessa porzione di memoria, ma i dati occupano la sua base e lo stack è allocato alla sua cima. Approfondiremo lo studio di questi registri nel Paragrafo C.3.1.

C.3 Memoria e indirizzamento

L'8088 ha un'organizzazione di memoria abbastanza grossolana a causa della combinazione infelice di una memoria di 1 MB e registri di 16 bit. La memoria di 1 MB richiede 20 bit per la rappresentazione di un suo indirizzo e dunque è impossibile memorizzare un puntatore alla memoria nei registri di 16 bit. Per aggirare il problema, la memoria è stata organizzata in segmenti di 64 KB e così è possibile rappresentare un indirizzo all'interno di un segmento con soli 16 bit. Vediamo l'architettura di memoria dell'8088 in maggior dettaglio.

C.3.1 Organizzazione di memoria e segmenti

La memoria dell'8088 consiste in una semplice successione di byte indirizzabili, contenente istruzioni, dati e stack. L'8088 usa il concetto di **segmento** per distinguere le porzioni di memoria che assolvono a compiti diversi. Ogni segmento dell'8088 è costituito da 65.536 byte consecutivi. Ne esistono quattro:

1. il segmento di codice
2. il segmento dati
3. il segmento di stack
4. il segmento extra.

Il primo contiene le istruzioni del programma. Il contenuto del registro PC viene interpretato sempre come un indirizzo di memoria appartenente a questo segmento. Il valore 0 del PC sta a indicare l'indirizzo minore del segmento di codice, *non* l'indirizzo zero della memoria. Il segmento dati contiene i dati, inizializzati e non, del programma. Quando BX è usato come puntatore si riferisce proprio a questo segmento. Il segmento di stack contiene le variabili locali e i risultati intermedi impilati sullo stack. Gli indirizzi di SP e BP appartengono sempre a questo segmento. Il segmento extra è un segmento di riserva che può essere allocato in memoria ovunque sia necessario.

A ciascuno di questi segmenti corrisponde un registro di segmento: si tratta dei registri di 16 bit CS, DS, SS ed ES. L'indirizzo iniziale di un segmento si ottiene estendendo i 16 bit del registro corrispondente concatenandogli quattro zeri sulla destra (nelle posizioni meno significative), producendo così un intero senza segno di 20 bit. Ciò vuol dire che i registri di segmento indicano sempre indirizzi multipli di 16 nello spazio degli indirizzi di 20 bit. Ogni registro di segmento punta alla base del proprio segmento. Gli indirizzi all'interno del segmento si costruiscono convertendo i 16 bit del registro nell'indirizzo fisico di 20 bit e sommandogli l'offset. Nella pratica, un indirizzo assoluto di memoria si calcola moltiplicando il registro di segmento per 16 e sommandogli poi l'offset. Per esempio, se DS vale 7 e BX vale 12, l'indirizzo indicato da BX è 7 × 16 + 12 = 124. In altre parole, l'indirizzo binario di 20 bit indotto da DS = 7 è 00000000000001110000. La somma dell'offset di 16 bit 0000000000001100 (12 in decimale) all'indirizzo base del segmento produce l'indirizzo fisico 00000000000001111100 (124 in decimale).

Ogni riferimento alla memoria richiede l'intervento di un registro di segmento per la costruzione dell'indirizzo fisico. Se un'istruzione contiene un indirizzo e non specifica alcun registro di segmento, allora è interpretata automaticamente come riferimento al segmento dati e si usa DS come base dell'offset. L'indirizzo fisico si ottiene sommando questa base all'indirizzo contenuto nell'istruzione. L'indirizzo fisico della successiva istruzione di codice si ottiene facendo scorrere il contenuto di CS di quattro posizioni a sinistra e sommando il valore del program counter. In altre parole, viene prima calcolato l'indirizzo di 20 bit indotto dai 16 bit del registro CS, poi viene effettuata la somma con i 16 bit del PC per formare l'indirizzo di memoria assoluto di 20 bit.

Il segmento di stack è costituito da parole di 2 byte, perciò il puntatore allo stack, SP, contiene sempre un numero pari. Lo stack viene riempito a partire dagli indirizzi più grandi in senso decrescente. L'istruzione PUSH decrementa il puntatore allo stack di 2 e memorizza l'operando nell'indirizzo di memoria calcolato mediante SS e SP. Il comando POP recupera il valore in cima e incrementa SP di 2. Gli indirizzi del segmento di stack al di sotto di quello indicato da SP si considerano liberi. La rimozione di una porzione dello stack si consegue con un semplice incremento di SP. Nella pratica DS e SS valgono sempre lo stesso valore, così si può usare un solo puntatore di 16 bit per accedere al segmento condiviso da stack e dati. Se DS e SS fossero stati diversi, si sarebbe reso necessario un diciassettesimo bit per distinguere tra puntatori al segmento dati e puntatori al segmento di stack. A posteriori, si può dire che la stessa definizione di un segmento di stack separato dai dati si è rivelata un errore.

Se gli indirizzi dei quattro segmenti sono molto distanti tra loro, allora i segmenti risultanti saranno disgiunti, ma ciò non è necessario; per esempio, se c'è poca memoria disponibile è possibile che i segmenti si sovrappongano almeno in parte. Al termine della com-

pilazione, la dimensione del codice del programma è nota. Una scelta efficiente prevede di far cominciare il segmento condiviso dallo stack e dai dati al primo multiplo di 16 che si trova dopo l'ultima istruzione. Questa scelta implica che i segmenti del codice e dei dati non useranno mai gli stessi indirizzi fisici.

C.3.2 Indirizzamento

Quasi tutte le istruzioni richiedono dati provenienti dalla memoria o dagli altri registri. L'8088 mette a disposizione un ventaglio di modalità d'indirizzamento abbastanza versatile per far riferimento ai dati. Molte istruzioni contengono due operandi, chiamati in genere **destinazione** e **sorgente**. Si pensi, per esempio, all'istruzione di copia o di somma:

```
MOV AX,BX   O   ADD CX,20
```

Il primo operando di queste istruzioni è la destinazione, il secondo la sorgente (l'ordine scelto da Intel è arbitrario: sarebbe stata lecita anche la scelta opposta). È evidente che la destinazione deve essere un **valore di sinistra dell'assegnamento** (*left value*), cioè deve essere una locazione in cui è lecito memorizzare un valore. Ciò implica che le costanti possono fungere da sorgenti, ma non da destinazioni.

Il progetto originale dell'8088 richiedeva che almeno un operando di ogni istruzione a due operandi fosse un registro. Questa scelta serviva per distinguere le **istruzioni di una parola** dalle **istruzioni di un byte** mediante il semplice controllo del registro indirizzato, a seconda che si trattasse di un **registro di una parola** o di uno **di un byte**. Nella prima versione del processore quest'idea veniva rispettata così rigorosamente tanto da non permettere neanche di impilare una costante sullo stack, dato che in questo caso nessuno dei due operandi coinvolge un registro. Le versioni successive sono meno rigorose, tuttavia l'idea ha continuato ugualmente a influenzare il progetto del processore. In alcuni casi uno dei due operandi viene omesso. Per esempio, nel caso dell'istruzione `MULB`, solo il registro `AX` può fungere da destinazione.

Esistono anche alcune istruzioni a un operando, come gli incrementi, gli scorrimenti, le negazioni, ecc. In questi casi non c'è vincolo sui registri e la differenza tra operazioni di una parola o di un byte deve essere desunta dall'opcode (cioè dal tipo d'istruzione).

L'8088 supporta quattro tipi di dati fondamentali: i **byte**, le **parole** di 2 byte, il tipo **long** di 4 byte e i **decimali binari** (*Binary Coded Decimals*, BCD), che impacchettano due cifre decimali all'interno di una parola. Quest'ultimo tipo non è supportato dall'interprete.

Un indirizzo di memoria si riferisce sempre a un byte, ma in caso di riferimento a una parola o a un dato long ci si riferisce implicitamente anche ai byte che seguono quello indicato. La parola all'indirizzo 20 comprende le locazioni di memoria 20 e 21. Il dato long all'indirizzo 24 occupa le locazioni da 24 a 27. L'8088 assume l'ordinamento **little endian**, cioè la parte meno significativa di una parola è memorizzata all'indirizzo minore. All'interno del segmento stack, le parole devono essere allocate in corrispondenza degli indirizzi pari. La coppia di registri `DX:AX` è l'unica messa a disposizione dal processore per i dati long (`AX` contiene la parola meno significativa).

La tabella della Figura C.3 fornisce una visione d'insieme delle modalità d'indirizzamento dell'8088. Diamone un breve resoconto. La prima riga della tabella elenca i registri utilizzabili come operandi di quasi tutte le istruzioni, come sorgente o come destinazione. Ci sono otto registri di una parola e otto di un byte.

Modalità	Operando	Esempi
Indirizzamento a registro		
Registro da un byte	Registro da un byte	AH, AL, BH, BL, CH, CL, DH, DL
Registro da una parola	Registro da una parola	AX, BX, CX, DX, SP, BP, SI, DI
Indirizzamento del segmento dati		
Indirizzo diretto	L'indirizzo segue l'opcode	(#)
A registro indiretto	L'indirizzo è nel registro	(SI), (DI), (BX)
A registro con spiazzamento	L'indirizzo è dato da registro + spiazz.	#(SI), #(DI), #(BX)
A registro con indice	L'indirizzo è BX + SI/DI	(BX)(SI), (BX)(DI)
A registro con indice e spiazzamento	BX + SI/DI + spiazzamento	#(BX)(SI), #(BX)(DI)
Indirizzamento del segmento di stack		
Indiretto a puntatore base	L'indirizzo è nel registro	(BP)
A puntatore base con spiazzamento	L'indirizzo è BP + spiazz.	#(BP)
A puntatore base con indice	L'indirizzo è BP + SI/DI	(BP)(SI), (BP)(DI)
A puntatore base con indice e spiazz.	BP + SI/DI + spiazzamento	#(BP)(SI), #(BP)(DI)
Dati immediati		
byte/parola immediato/a	I dati sono parte dell'istruzione	#
Indirizzamento implicito		
Istruzione di push/pop	Indirizzo indiretto (SP)	PUSH, POP, PUSHF, POPF
Flag di load/store	registro dei flag di stato	LAHF, STC, CLC, CMC
Traduzione XLAT	AL, BX	XLAT
Istruzioni reiterative su stringhe	(SI), (DI), (CX)	MOVS, CMPS, SCAS
Istruzioni di input/out	AX, AL	IN #, OUT #
Conversioni di byte o parole	AL, AX, DX	CBW, CWD

Figura C.3 Modalità d'indirizzamento degli operandi. Il simbolo # indica un valore numerico o un'etichetta.

La seconda riga riguarda le modalità d'indirizzamento del segmento dati. Questi indirizzi sono sempre racchiusi tra parentesi a indicare che il contenuto va inteso come indirizzo e non come valore. La **modalità d'indirizzamento** più semplice è quella **diretta**, mediante cui l'istruzione contiene l'indirizzo dei dati nell'operando stesso. Per esempio:

```
ADD CX,(20)
```

specifica che bisogna sommare a `CX` il contenuto della parola di memoria agli indirizzi 20 e 21. Nel linguaggio assemblativo si usa spesso rappresentare le locazioni di memoria mediante etichette e non con il loro valore numerico; la conversione da etichette a numeri viene effettuata dall'assemblatore. Anche la destinazione delle istruzioni `CALL` e `JMP` può essere specificata da un'etichetta. Le parentesi che racchiudono l'etichetta sono fondamentali (per l'assemblatore che stiamo descrivendo) perché anche

```
ADD CX,20
```

è un'istruzione valida, ma produce la somma di `CX` con la costante 20, non con il contenuto della locazione 20. Il simbolo # usato nella Figura C.3 indica le costanti numeriche, le etichette o le espressioni costanti contenenti un'etichetta.

Nell'**indirizzamento a registro indiretto**, l'indirizzo dell'operando viene memorizzato in uno dei registri `BX`, `SI` o `DI`. In tutti e tre i casi si tratta di un operando che fa riferimento al segmento dati. È possibile anche far precedere un registro da un simbolo di costante, nel qual caso l'indirizzo si ottiene sommando la costante al registro. Questo tipo d'in-

dirizzamento si chiama **indirizzamento con spiazzamento** ed è particolarmente utile per il trattamento degli array. Per esempio, se SI contiene il valore 5, è possibile caricare in AL il quinto carattere della stringa *FORMAT* tramite

```
MOVB AL,FORMAT(SI).
```

L'intera stringa può essere scandita incrementando o decrementando il registro a ogni passo. Se si usano gli operandi di una parola, bisogna avere l'accortezza di modificare il registro di 2 unità alla volta.

È possibile anche memorizzare la base dell'array (cioè il suo indirizzo minore) nel registro BX e utilizzare il registro SI o DI per scandirlo. Si parla in tal caso d'**indirizzamento a registro con indice**. Per esempio

```
PUSH (BX)(DI)
```

preleva il contenuto della locazione del segmento dati il cui indirizzo è la somma dei registri BX e DI, e lo impila in cima allo stack. Gli ultimi due tipi d'indirizzamento possono essere combinati per ottenere l'**indirizzamento a registro con indice e spiazzamento**, come in

```
NOT 20(BX)(DI)
```

che fa il complemento della parola di memoria che si trova agli indirizzi BX + DI + 20 e seguente.

Tutte le modalità d'indirizzamento indiretto del segmento dati si possono usare anche per il segmento di stack, ma in tal caso si usa come puntatore base BP al posto del registro BX. Così facendo, (BP) è la sola modalità d'indirizzamento a registro indiretto disponibile per lo stack, ma esistono altre modalità via via più complicate che culminano nella modalità indiretta a puntatore base con indice e spiazzamento, per esempio -1(BP)(SI). Queste modalità sono utili per indirizzare le variabili locali e i parametri di funzione, memorizzati presso gli indirizzi di stack delle subroutine. Questo sistema è descritto più a fondo nel Paragrafo C.4.5.

Tutti gli indirizzi che rispettano le modalità d'indirizzamento fin qui trattate possono fungere da sorgenti o da destinazioni delle operazioni e si dicono **indirizzi effettivi**. Le modalità d'indirizzamento delle rimanenti due righe non possono essere usate per le destinazioni e non danno luogo a indirizzi effettivi. Possono servire solo per specificare sorgenti.

La modalità d'indirizzamento in cui l'operando dell'istruzione è una costante di un byte o di una parola si dice **indirizzamento immediato**. Per esempio

```
CMP AX,50
```

confronta AX con la costante 50 e memorizza l'esito del confronto nel registro dei flag.

Infine, alcune istruzioni usano l'**indirizzamento implicito**, secondo il quale l'operando o gli operandi dell'istruzione sono specificati implicitamente dall'istruzione stessa. Per esempio l'istruzione

```
PUSH AX
```

impila il contenuto di AX in cima allo stack mediante il decremento di SP e la copia di AX nella nuova locazione puntata da SP. Il registro puntatore allo stack non viene indicato dall'istruzione, ma il solo fatto che si tratti di un'istruzione PUSH implica l'uso di SP. Anche le istruzioni di manipolazione dei flag usano implicitamente il registro dei flag senza nominarlo, e molte altre istruzioni usano operandi impliciti.

L'8088 dispone d'istruzioni speciali per il trasferimento (MOVS), il confronto (CMPS) e la scansione (SCAS) di stringhe. Queste istruzioni sulle stringhe comportano la modifica automatica dei registri indice SI e DI al termine delle operazioni corrispondenti. Questo comportamento si dice **autoincremento** o **autodecremento**. SI e DI vengono modificati a seconda del valore assunto dal **flag della modalità d'incremento** (*direction flag*) contenuto nel registro dei flag: se vale 0 allora si ha un incremento, altrimenti un decremento. La modifica è di un'unità alla volta nel caso d'istruzioni di un byte e di 2 per le istruzioni di una parola. In un certo senso anche il puntatore allo stack segue le modalità di autoincremento e autodecremento: viene decrementato di 2 all'inizio di una PUSH e incrementato di 2 alla fine di una POP.

C.4 Istruzioni dell'8088

Il fulcro di ogni calcolatore è l'insieme d'istruzioni che può eseguire. La comprensione profonda di un calcolatore passa per la comprensione dell'insieme delle sue istruzioni. I paragrafi seguenti sono dedicati all'esposizione delle istruzioni più importanti dell'8088. La Figura C.4 ne mostra alcune.

C.4.1 Trasferimento, copia e aritmetica

Il primo gruppo contiene le istruzioni di copia e di trasferimento. L'istruzione di gran lunga più utilizzata è MOV, che specifica esplicitamente una sorgente e una destinazione. Se la sorgente è un registro, allora la destinazione può essere un indirizzo effettivo. Nella tabella delle istruzioni indichiamo con una *r* gli operandi registro e con una *e* gli indirizzi effettivi, dunque la precedente combinazione degli operandi della MOV si indica con $e \leftarrow r$. Il verso della freccia segue la convenzione sintattica secondo cui si indicano prima le destinazioni e poi le sorgenti; dunque $e \leftarrow r$ vuol dire che un registro viene copiato in un indirizzo effettivo.

Si tratta solo del primo elemento della colonna *Operandi* in corrispondenza dell'istruzione MOV, ma è anche possibile avere per sorgente un indirizzo effettivo e per destinazione un registro, $r \leftarrow e$, oppure un dato immediato come sorgente e un indirizzo effettivo come destinazione, che indichiamo con $e \leftarrow \#$. I dati immediati sono contrassegnati nella tabella con il *cancelletto* (#). La *B* indicata tra parentesi dopo il nome dell'istruzione serve a ricordare che esiste sia l'istruzione MOV per i trasferimenti di parole, sia l'istruzione MOVB per i trasferimenti di byte. Perciò la prima riga della tabella riguarda in realtà sei istruzioni diverse.

Le istruzioni di trasferimento non hanno alcun effetto sul registro dei codici di condizione e quindi le ultime quattro colonne sono contrassegnate con il simbolo "-". Si noti che in realtà le istruzioni di trasferimento non trasferiscono i dati, ma li copiano, ovvero la sorgente non viene modificata come accadrebbe a seguito di un vero trasferimento.

La seconda istruzione della tabella è XCHG, che scambia il contenuto di un registro con il contenuto di un indirizzo effettivo. Il simbolo per lo scambio è $\leftrightarrow$. Anche in questo caso esiste sia la versione per le parole, sia quella per i byte. La riga della tabella contiene quindi $r \leftrightarrow e$ nella colonna *Operandi* in corrispondenza della riga di XCHG. L'istruzione successiva è LEA, che sta per Load Effective Address ("caricamento di un indirizzo effettivo"). Serve a calcolare il valore numerico dell'indirizzo effettivo e a memorizzarlo in un registro.

Abbreviazione	Descrizione	Operandi	Flag di stato			
			O	S	Z	C
MOV(B)	Trasferimento di parola o byte	r ← e, e ← r, e ← #	-	-	-	-
XCHG(B)	Scambio di parole	r ↔ e	-	-	-	-
LEA	Caricamento di un indirizzo fisico	r ← #e	-	-	-	-
PUSH	Push sullo stack	e, #	-	-	-	-
POP	Pop dallo stack	e	-	-	-	-
PUSHF	Push dei flag	-	-	-	-	-
POPF	Pop dei flag	-	-	-	-	-
XLAT	Traduzione di AL	-	-	-	-	-
ADD(B)	Somma di parole	r ← e, e ← r, e ← #	*	*	*	*
ADC(B)	Somma di parole con riporto	r ← e, e ← r, e ← #	*	*	*	*
SUB(B)	Sottrazione di parole	r ← e, e ← r, e ← #	*	*	*	*
SBB(B)	Sottrazione di parole con prestito	r ← e, e ← r, e ← #	*	*	*	*
IMUL(B)	Moltiplicazione con segno	e	*	U	U	*
MUL(B)	Moltiplicazione senza segno	e	*	U	U	*
IDIV(B)	Divisione con segno	e	U	U	U	U
DIV(B)	Divisione senza segno	e	U	U	U	U
CBW	Estensione da byte a parola	-	-	-	-	-
CWD	Estensione da parola a double	-	-	-	-	-
NEG(B)	Complemento binario	e	*	*	*	*
NOT(B)	Complemento logico	e	-	-	-	-
INC(B)	Incremento della destinazione	e	*	*	*	-
DEC(B)	Decremento della destinazione	e	*	*	*	-
AND(B)	AND	e ← r, r ← e, e ← #	0	*	*	0
OR(B)	OR	e ← r, r ← e, e ← #	0	*	*	0
XOR(B)	OR esclusivo	e ← r, r ← e, e ← #	0	*	*	0
SHR(B)	Scorrimento logico verso destra	e ← 1, e ← CL	*	*	*	*
SAR(B)	Scorrimento aritmetico verso destra	e ← 1, e ← CL	*	*	*	*
SAL(B) (=SHL(B))	Scorrimento logico verso sinistra	e ← 1, e ← CL	*	*	*	*
ROL(B)	Rotazione a sinistra	e ← 1, e ← CL	*	-	-	*
ROR(B)	Rotazione a destra	e ← 1, e ← CL	*	-	-	*
RCL(B)	Rotazione a sinistra con riporto	e ← 1, e ← CL	*	-	-	*
RCR(B)	Rotazione a destra con riporto	e ← 1, e ← CL	*	-	-	*
TEST(B)	Test degli operandi	e ↔ r, e ↔ #	0	*	*	0
CMP(B)	Confronto degli operandi	e ↔ r, e ↔ #	*	*	*	*
STD	Asserzione flag modalità di incr. (↓)	-	-	-	-	-
CLD	Azzeramento flag modalità di incr. (↑)	-	-	-	-	-
STC	Asserzione flag di riporto	-	-	-	-	1
CLC	Azzeramento flag di riporto	-	-	-	-	0
CMC	Complemento del riporto	-	-	-	-	*
LOOP	Salta all'indietro se decrementato CX ≥ 0	etichetta	-	-	-	-
LOOPZ LOOPE	Salta se Z=1 e DEC(CX) ≥ 0	etichetta	-	-	-	-
LOOPNZ LOOPNE	Salta se Z=0 e DEC(CX) ≥ 0	etichetta	-	-	-	-
REP REPZ REPNZ	Istruzione ripetitiva sulle stringhe	istruzioni sulle stringhe	-	-	-	-
MOVS(B)	Trasf. di una stringa di una parola	-	-	-	-	-
LODS(B)	Caricam. di una stringa di parole	-	-	-	-	-
STOS(B)	Memorizz. di una stringa di parole	-	-	-	-	-
SCAS(B)	Scansione di una stringa di parole	-	*	*	*	*
CMPS(B)	Confronto tra stringhe di parole	-	*	*	*	*
JCC	Salto condizionato	etichetta	-	-	-	-
JMP	Salto all'etichetta	e, etichetta	-	-	-	-
CALL	Chiamata di subroutine	e, etichetta	-	-	-	-
RET	Ritorno da subroutine	-, #	-	-	-	-
SYS	Chiamata di sistema per trap	-	-	-	-	-

Figura C.4 Alcune delle istruzioni più importanti dell'8088.

Segue `PUSH` che impila operandi sullo stack. L'operando esplicito può essere una costante (# nella colonna *Operandi*) o un indirizzo effettivo (*e* nella colonna *Operandi*). C'è anche un operando implicito, `SP`, che non si evince dalla sintassi dell'istruzione. La semantica dell'istruzione è invece la seguente: decrementa `SP` di 2 e memorizza l'operando esplicito nella nuova locazione di memoria puntata da `SP`.

Troviamo poi l'istruzione `POP`, che rimuove un operando dallo stack e lo salva in un indirizzo effettivo. Anche le due istruzioni successive, `PUSHF` e `POPF`, hanno operandi impliciti, dato che effettuano il push e la pop del registro dei flag. Lo stesso vale per `XLAT`, che serve a caricare nel registro `AL` il byte che si trova all'indirizzo `AL + BX`. È un'istruzione molto utile per fare ricerche rapide nelle tabelle di dimensioni pari a 256 byte.

Le istruzioni `IN` e `OUT` sono definite dall'8088, ma non sono state implementate nell'interprete (e non sono elencate nelle Figura C.4). Si tratta infatti d'istruzioni di trasferimento da e verso i dispositivi di I/O. L'indirizzo implicito è sempre il registro `AX` e il secondo operando dell'istruzione è il numero di porta del registro di dispositivo desiderato.

Il secondo gruppo della tabella comprende le istruzioni di somma e sottrazione. Ciascuna di loro dispone delle stesse tre combinazioni di operandi di `MOV`: da indirizzo effettivo a registro, da registro a indirizzo effettivo e da costante a indirizzo effettivo. Quindi la colonna *Operandi* della tabella contiene $r \leftarrow e$, $e \leftarrow r$ e $e \leftarrow \#$. Tutte le istruzioni di questo gruppo modificano i flag di stato (cioè i bit di overflow *O*, di segno *S*, di zero *Z* e di riporto *C*) in base al risultato dell'istruzione eseguita. Per esempio, *O* viene asserito se il risultato non può essere espresso correttamente con il numero di bit consentiti, e viene azzerato altrimenti. Se si prende il numero più grande rappresentabile con 16 bit, 0x7fff (32.767 in decimale), e lo si somma a se stesso, il risultato non può essere espresso con un numero di 16 bit con segno, così *O* viene asserito a indicare l'errore. Gli altri flag di stato sono gestiti in modo analogo. Indichiamo con un asterisco (*) i flag che sono modificati dalle istruzioni. Le istruzioni `ADC` e `SBB` attingono al flag di riporto e lo usano come bit di riporto o di prestito generato dall'operazione precedente. Questa caratteristica è particolarmente utile per rappresentare gli interi di 32 bit (o ancora più lunghi) usando più parole consecutive. Tutte le istruzioni di somma o sottrazione esistono anche in versione per byte.

Il gruppo successivo contiene le istruzioni di moltiplicazione e divisione `IMUL` e `IDIV` con operandi interi con segno, e `MUL` e `DIV` per quelli senza segno. Le versioni per byte definiscono implicitamente come destinazione i registri accoppiati `AH:AL`, mentre le versioni per operandi di una parola usano come destinazione implicita la coppia di registri `DX:AX`. `DX` o `AH` vengono riscritti anche se il risultato della moltiplicazione occupa solo una parola o un byte. I bit di overflow e di riporto sono asseriti quando il risultato eccede la prima parola o byte; comunque la moltiplicazione può sempre andare a buon fine perché la destinazione (due parole o due byte) è abbastanza capiente per contenere il risultato. Il contenuto del flag *Z* e *S* è indefinito (U) dopo una moltiplicazione.

Anche la divisione usa come destinazione le combinazioni di registri `DX:AX` (per il resto) `AH:AL` (per il quoziente). Al termine di una divisione è indefinito il contenuto dei flag di resto, di overflow, di segno, e di zero. L'operazione causa una **trap** se il divisore è nullo o se non c'è abbastanza spazio per rappresentare il quoziente; la trap causa la terminazione del programma a meno che sia stata definita una routine per la gestione delle trap. Inoltre, conviene gestire via software il segno prima e dopo la divisione, perché l'8088 definisce il segno del resto in base a quello del dividendo, mentre dal punto di vista matematico il resto è sempre non negativo.

Le istruzioni AAA (ASCII Adjust for Addition) e DAA (Decimal Ajust for Addition) per le operazioni sui BCD non sono state implementate nell'interprete e quindi non sono presenti nella tabella.

C.4.2 Operazioni logiche, su bit e di scorrimento

Il blocco d'istruzioni seguente raggruppa le estensioni con segno, le negazioni, il complemento logico, l'incremento e il decremento. Le operazioni d'estensione con segno non hanno operandi espliciti, bensì operano sulle combinazioni di registri DX:AX o AH:AL. L'operando delle altre istruzioni può trovarsi invece presso qualsiasi indirizzo effettivo. Le operazioni NEG, INC e DEC hanno effetto sui flag in accordo con la loro semantica, anche se le operazioni d'incremento e decremento non modificano il bit di riporto e ciò è considerato da molti un errore di progettazione.

Seguono le istruzioni del gruppo logico, ciascuna dotata di due operandi e fedele alla semantica dell'operazione logica corrispondente. Il gruppo degli scorrimenti e delle rotazioni contiene istruzioni che hanno un indirizzo effettivo come destinazione, ma la loro sorgente può essere il registro di un byte CL o il numero 1. Gli scorrimenti modificano i quattro flag, mentre le rotazioni agiscono solo sui bit di riporto e di overflow. Il bit di riporto contiene sempre il bit che, a seguito di una rotazione o di uno scorrimento, "cade" fuori dal registro. Nelle rotazioni con riporto RCR, RCL, RCRB e RCLB, il bit di riporto e l'operando che si trova all'indirizzo effettivo costituiscono insieme una combinazione di 17 o di 9 bit per lo scorrimento circolare di registro, il che facilita gli scorrimenti e le rotazioni di più parole alla volta.

Il gruppo successivo d'istruzioni serve alla manipolazione dei bit dei flag. Queste operazioni sono utili soprattutto per la preparazione dei salti condizionati. Il rapporto tra i due operandi delle istruzioni di test e confronto, che non vengono modificati da queste operazioni, è indicato con il simbolo di doppia freccia (↔). L'operazione TEST comporta il calcolo dell'AND dei due operandi e, in base al risultato, imposta il bit zero e il bit di segno. Il risultato dell'AND non viene memorizzato da nessuna parte e gli operandi non vengono modificati. L'istruzione CMP invece calcola la differenza dei due operandi e modifica tutti i flag, in base al risultato del confronto. Il flag della modalità d'incremento, che specifica se le istruzioni sulle stringhe debbano incrementare o decrementare SI e DI, può essere asserito o azzerato dalle istruzioni STD e CLD rispettivamente.

L'8088 ha anche un **flag di parità** e uno **di riporto ausiliario**. Il primo indica la parità del risultato, e il secondo se si è verificato un overflow nel mezzo byte (4 bit) meno significativo della destinazione. Esistono anche le istruzioni LAHF e SAHF per la copia in AH del byte meno significativo del registro di flag o viceversa. Il flag di overflow si trova nel byte più significativo del registro dei codici di condizione e perciò non è interessato da queste istruzioni. Questo tipo d'istruzioni e i flag sono usati per lo più per la retrocompatibilità con i processori 8080 e 8085.

C.4.3 Cicli e operazioni iterative su stringhe

Scorrendo la tabella incontriamo poi le istruzioni di ciclo. L'istruzione LOOP decrementa il registro CX e, se il risultato è positivo, salta all'etichetta specificata. Anche le istruzioni LOOPZ, LOOPE, LOOPNZ e LOOPNE confrontano CX con *Z* per decidere se interrompere l'esecuzione del ciclo.

La destinazione di tutte le istruzioni LOOP deve trovarsi entro 128 byte dalla posizione corrente del program counter, perché l'istruzione contiene solo un offset di 8 bit con segno. Non si può calcolare con esattezza il numero d'*istruzioni* che è possibile saltare con 128 byte perché possono avere lunghezze differenti. In genere il primo byte delle istruzioni specifica il loro tipo, e alcune istruzioni occupano un solo byte nel segmento di codice. Spesso il secondo byte serve a definire i registri e le loro modalità e, se l'istruzione contiene spiazzamenti o dati immediati, la sua lunghezza può arrivare fino a quattro o sei byte. La lunghezza media delle istruzioni è di circa 2,5 byte, perciò LOOP consente in genere di saltare non più di 50 istruzioni in avanti o all'indietro.

Esistono anche meccanismi iterativi speciali per il trattamento delle stringhe, resi disponibili dalle istruzioni REP, REPZ e REPNZ. Anche le cinque istruzioni successive nella Figura C.4 definiscono gli operandi implicitamente e usano le modalità di autoincremento o di autodecremento dei registri indice. Tutte queste istruzioni prevedono che il registro indice SI punti al **segmento dati**, mentre DI punta al **segmento extra** localizzato all'indirizzo specificato da ES. L'istruzione MOVSB può essere usata insieme a REP per trasferire un'intera stringa in una sola istruzione. La lunghezza della stringa è contenuta nel registro CX. Dal momento che MOVSB non modifica i flag, non è possibile usare REPNZ durante la copia per verificare se un certo byte è equivalente al codice ASCII del carattere zero (carattere di terminazione delle stringhe), ma si può ovviare a questo problema usando REPNZ SCASB per memorizzare in CX la posizione del byte zero e poi usare questo valore in REP MOVSB per la copia. Questo concetto verrà chiarito dall'esempio di codice sulle stringhe esposto nel Paragrafo C.8. Nell'uso di queste istruzioni bisognerebbe prestare particolare attenzione al registro di segmento ES, a meno che ES e DS non assumano lo stesso valore. L'interprete si avvale di un modello piccolo di memoria e dunque impone ES = DS = SS.

C.4.4 Istruzioni di salto e di chiamata

L'ultimo gruppo d'istruzioni contiene i salti condizionati e non, le chiamate di subroutine e quelle di ritorno. L'operazione più semplice è svolta dall'istruzione JMP che può contenere come destinazione del salto un'etichetta o un qualsiasi indirizzo effettivo. Esistono due tipi di salto: **corto** e **lungo** (*near* e *far jump*). Si ha un salto corto quando la destinazione si trova nel segmento di codice corrente, che non può cambiare durante l'esecuzione dell'operazione. In caso di salto lungo invece il salto modifica il contenuto del registro CS. Nella versione con etichetta, il nuovo valore del registro del segmento di codice è contenuto nell'istruzione appena dopo l'etichetta, mentre nella versione con indirizzo effettivo si effettua il fetch di un dato long dalla memoria e lo si scompone in due: la parola meno significativa corrisponde all'etichetta della destinazione, quella più significativa contiene il nuovo valore del registro del segmento di codice.

Naturalmente questa distinzione non sorprende affatto: per spostarsi in uno spazio degli indirizzi di 20 bit è necessario prevedere qualche meccanismo per poter utilizzare più di 16 bit. Il procedimento di salto termina con l'attribuzione di nuovi valori a CS e a PC.

Salti condizionati

L'8088 ha 15 istruzioni di salto condizionato, alcune delle quali hanno più di un nome (per esempio JGE, *Jump Greater or Equal*, è la stessa istruzione di JNL, *Jump Not Less than*); sono tutte elencate nella Figura C.5. Queste istruzioni consentono salti di al massimo 128 byte a

Istruzione	Descrizione	Condizione del salto
JNA, JBE	Al di sotto o uguale	CF=1 or ZF=1
JNB, JAE, JNC	Non al di sotto	CF=0
JE, JZ	Zero, uguale	ZF=1
JNLE, JG	Maggiore di	SF=OF and ZF=0
JGE, JNL	Maggiore o uguale	SF=OF
JO	Overflow	OF=1
JS	Segno negativo	SF=1
JCXZ	CX vale zero	CX=0
JB, JNAE, JC	Al di sotto	CF=1
JNBE, JA	Al di sopra	CF=0&ZF=0
JNE, JNZ	Non zero, diverso	ZF=0
JL, JNGE	Minore di	SF≠OF
JLE, JNG	Minore o uguale	SF≠OF or ZF=1
JNO	No overflow	OF=0
JNS	Non negativo	SF=0

Figura C.5 Salti condizionati.

partire dall'istruzione. Se la destinazione è più distante di questo valore bisogna ricorrere a un meccanismo di costruzione di salto: si usa il salto con condizione opposta e lo si fa seguire da un salto incondizionato alla destinazione desiderata. L'effetto della combinazione di queste due istruzioni è proprio il salto a lungo raggio che si voleva ottenere. Per esempio, invece di scrivere

```
JB FARLABEL
```

scriviamo

```
    JNA 1f
    JMP FARLABEL
1:
```

Dunque, se non è possibile effettuare una `JB` (*Jump Below*) verso l'etichetta lontana *FARLABEL* si usa una `JNA` (*Jump Not Above*) che specifica un'etichetta vicina *1* e la si fa seguire da un salto incondizionato all'etichetta lontana *FARLABEL*. L'effetto è lo stesso, ma richiede un consumo di risorse in tempo e spazio leggermente maggiore. Questo meccanismo complica un po' il calcolo delle distanze dei salti. Se una distanza è di poco minore di 128, ma alcune delle istruzioni frapposte tra il salto e la destinazione sono a loro volta salti condizionati, non è possibile stabilire la natura del salto (corto o lungo) finché non sono stati valutati i salti che si trovano nel mezzo. Per sicurezza, in questi casi l'assemblatore adotta un atteggiamento prudente e alle volte costruisce un salto in due istruzioni anche se non è strettamente necessario. Viceversa predilige il salto condizionato diretto solo quando è assolutamente certo che la destinazione si trovi alla portata di un salto corto.

Molti salti condizionati dipendono dai flag di stato e sono preceduti da un'istruzione di test o di confronto. Le istruzioni `CMP` sottraggono la sorgente dall'operando destinazione, impostano i codici di condizione e scartano il risultato. Nessuno degli operandi risulta modificato da queste istruzioni. Se il risultato è zero, o se il suo bit di segno vale 1 (cioè è negativo), il flag corrispondente viene asserito. Se il risultato non può essere espresso con il numero di bit disponibili, viene asserito il flag di overflow. Se si verifica un riporto al di là del bit più significativo, il flag di riporto si attiva. I salti condizionati possono dipendere da diverse combinazioni di questi bit.

Se gli operandi sono interi con segno bisogna usare `JG` (*Jump Greater than*) o `JL` (*Jump Less than*), mentre se sono senza segno si usano `JA` (*Jump Above*) e `JB` (*Jump Below*).

C.4.5 Chiamate di subroutine

L'8088 dispone di un'istruzione per la chiamata di procedure che, nella terminologia del linguaggio assemblativo, si chiamano **subroutine**. Analogamente al caso delle istruzioni di salto, anche per questo tipo d'istruzioni si distinguono le **chiamate ravvicinate** e quelle **a distanza** (*near* e *far call*). Solo le prime sono state implementate nell'interprete. La destinazione può essere un'etichetta o può trovarsi presso un indirizzo effettivo. I parametri necessari alla subroutine devono essere impilati sullo stack in ordine inverso, come illustrato nella Figura C.6. Nella terminologia del linguaggio assemblativo i parametri si chiamano **argomenti**, ma i due termini sono interscambiabili. Dopo il push degli argomenti si esegue l'istruzione `CALL`, che comincia con l'impilare sullo stack il valore corrente del program counter per salvare l'indirizzo di ritorno (quello dal quale bisogna riprendere l'esecuzione della routine chiamante al termine della subroutine).

A questo punto viene caricato il nuovo program counter a partire dall'etichetta o dall'indirizzo effettivo. Se la chiamata è a distanza, il registro del segmento di codice (`CS`) viene impilato prima del program counter (`PC`) e si procede al caricamento di entrambi a partire dai dati immediati o dall'indirizzo effettivo. Questo caricamento conclude l'esecuzione dell'istruzione `CALL`.

L'istruzione di ritorno, `RET`, non fa altro che recuperare l'indirizzo di ritorno dalla cima dello stack e lo scrive nel program counter, così l'esecuzione del programma riprende immediatamente dall'istruzione successiva alla `CALL`. Alle volte l'istruzione `RET` contiene un

Figura C.6 Un esempio di stack durante l'esecuzione di una subroutine.

numero positivo come dato immediato che rappresenta il numero di byte occupati dagli argomenti impilati sullo stack prima della chiamata; basta sommare questo numero a SP per rimuovere gli argomenti dallo stack. La variante a distanza RETF esegue la pop del registro del segmento di codice dopo la pop del program counter, proprio come ci si poteva aspettare.

Gli argomenti devono essere accessibili alla subroutine, perciò in genere la sua esecuzione comincia con il push del puntatore base e una copia del valore corrente di SP in BP. Così facendo, il puntatore base punta ora al suo valore precedente e l'indirizzo di ritorno si trova alla posizione BP + 2, mentre il primo e il secondo argomento si trovano rispettivamente alla posizione BP + 4 e BP + 6. Se la procedura ha delle variabili locali allora è possibile sottrarre il numero di byte richiesto dal valore del puntatore allo stack e le variabili possono essere indirizzate a partire dal puntatore base sommandogli offset negativi. Nell'esempio della Figura C.6 ci sono tre variabili locali di una parola ciascuna, localizzate agli indirizzi BP – 2, BP – 4 e BP – 6. Dunque il registro BP utilizzabile per raggiungere l'intero insieme di argomenti e variabili locali della procedura corrente.

Lo stack viene usato nel modo consueto per contenere i risultati intermedi o per preparare gli argomenti per la chiamata successiva. Al ritorno da una subroutine lo stack può essere ripristinato senza calcolare lo spazio da essa utilizzato, ma basterà copiare il puntatore base nel puntatore allo stack, effettuare la pop del vecchio BP e infine eseguire l'istruzione RET.

Durante una chiamata di subroutine i valori di alcuni registri possono venir modificati. È buona pratica usare qualche tipo di convenzione, di modo che la routine chiamante non debba preoccuparsi dei registri usati dalla routine chiamata. La soluzione più semplice è adottare la stessa convenzione che esiste tra le chiamate di sistema e le subroutine ordinarie: la routine chiamata può modificare solo i registri AX e DX, perciò se uno di questi contiene informazioni importanti per il chiamante è consigliabile che questi ne salvi il contenuto nello stack prima d'impilare gli argomenti della chiamata. Se la subroutine usa anche altri registri allora può impilarli sullo stack immediatamente all'inizio della sua esecuzione e poi ripristinarli prima di chiamare l'istruzione RET. In breve, una buona convenzione richiede che il chiamante salvi AX e DX se li ritiene importanti, mentre il chiamato salva ogni altro registro che intende modificare.

C.4.6 Chiamate di sistema e subroutine di sistema

Per separare le operazioni su file (apertura, chiusura, lettura e scrittura) dalla programmazione in linguaggio assemblativo, i programmi sono eseguiti a livello superiore rispetto a quello del sistema operativo. L'interprete è stato dotato di sette chiamate di sistema e cinque funzioni (elencate nella Figura C.7) che ne permettono l'esecuzione su diverse piattaforme hardware.

Queste routine possono essere attivate con la sequenza di chiamata standard: si comincia con il push degli argomenti sullo stack in ordine inverso, si impila il numero di chiamata e infine si esegue l'istruzione SYS senza operandi per la trap di sistema. La routine di sistema reperisce nello stack tutte le informazioni che le sono necessarie, compreso il numero di chiamata e il servizio di sistema richiesto. I risultati sono restituiti nel registro AX o nella combinazione di registri DX:AX (se il risultato è di tipo long).

L'istruzione SYS non modifica i valori degli altri registri. Al suo completamento, gli argomenti restano impilati sullo stack e vanno rimossi (a cura del chiamante) agendo sul puntatore allo stack, a meno che non servano per una chiamata successiva.

No.	Nome	Argomenti	Valore restituito	Descrizione
5	_OPEN	*name, 0/1/2	descrittore di file	Apre un file
8	_CREAT	*name, *mode	descrittore di file	Crea un file
3	_READ	fd, buf, nbytes	# di byte	Legge n byte nel buffer buf
4	_WRITE	fd, buf, nbytes	# di byte	Scrive n byte nel buffer buf
6	_CLOSE	fd	0 se ha successo	Chiude il file con descrittore fd
19	_LSEEK	fd, offset(long), 0/1/2	posizione (long)	Sposta il puntatore del file
1	_EXIT	status		Interrompe un processo
117	_GETCHAR		carattere letto	Legge un carattere da standard input
122	_PUTCHAR	char	byte scritto	Scrive un carattere sullo standard output
127	_PRINTF	*format, arg		Stampa formattata sullo standard output
121	_SPRINTF	buf, *format, arg		Stampa formattata nel buffer buf
125	_SSCANF	buf, *format, arg		Legge gli argomenti dal buffer buf

Figura C.7 Alcune chiamate di sistema UNIX e alcune subroutine implementate nell'interprete.

Per comodità, è possibile definire all'inizio del programma assemblativo alcune costanti per i nomi delle chiamate di sistema, in modo da poterle richiamare per nome invece che per numero. Gli esempi che seguono ci permetteranno di descrivere in dettaglio molte chiamate di sistema, perciò in questo paragrafo ne diamo un resoconto abbreviato.

L'apertura dei file si effettua con le chiamate OPEN e CREAT, che prendono come primo argomento l'indirizzo della stringa contenente il nome del file. Il secondo argomento di OPEN vale 0 (se il file va aperto in lettura), 1 (se in scrittura) o 2 (lettura/scrittura). Se il file è scrivibile, ma non esiste, allora la chiamata provvede a crearlo. La chiamata CREAT produce un file vuoto con i permessi specificati dal secondo argomento. OPEN e CREAT restituiscono entrambe un piccolo intero nel registro AX; questo numero si chiama **descrittore di file** e può essere usato per lettura, scrittura o chiusura del file. Un descrittore di file negativo indica il fallimento della chiamata. All'avvio di un programma vengono aperti automaticamente tre file con descrittori 0 (standard input), 1 (standard output) e 2 (standard error output).

Le chiamate READ e WRITE hanno tre argomenti: il descrittore di file, un buffer per i dati e il numero di byte da trasferire. Dal momento che gli argomenti sono impilati in ordine inverso, si effettua prima il push del numero di byte, e poi i push dell'indirizzo del buffer, del descrittore di file e infine del numero di chiamata (READ o WRITE). Quest'ordine rispetta la sequenza degli argomenti della chiamata C standard

```
read(fd, buffer, bytes);
```

i cui parametri sono impilati nell'ordine *bytes*, *buffer* e *fd*.

La chiamata CLOSE richiede solo il descrittore di file e restituisce 0 in AX se la chiusura del file ha avuto successo. La chiamata EXIT richiede di impilare sullo stack lo stato di uscita del processo e non restituisce alcun valore.

La chiamata LSEEK modifica il **puntatore di lettura/scrittura** di un file già aperto. Il primo argomento è il descrittore di file, e il secondo è un dato long, perciò va scomposto nelle due parole che lo costituiscono e va impilata per prima la metà più significativa, anche nel caso in cui l'offset fosse contenuto tutto nella metà meno significativa. Il terzo argomento indica se il puntatore di lettura/scrittura va calcolato relativamente all'inizio del file (caso 0), alla posizione corrente (caso 1) o alla fine del file (caso 2). Il valore restituito

è la nuova posizione del puntatore relativa all'inizio del file, memorizzato come long nei registri DX:AX.

Veniamo ora alle funzioni che non sono chiamate di sistema. La funzione GETCHAR legge un carattere dall'input standard, lo salva in AL e azzera AH. In caso di fallimento AX è posto invece a −1. La chiamata PUTCHAR scrive un byte sullo standard output e, in caso di successo, restituisce lo stesso byte scritto, mentre restituisce −1 in caso di fallimento.

La chiamata PRINTF stampa informazioni formattate. Il primo argomento della chiamata è l'indirizzo della stringa di formattazione, che specifica il formato dell'output. La stringa "%d" indica che l'argomento successivo è un intero presente nello stack e che quindi va convertito in notazione decimale prima della stampa. Allo stesso modo, "%x" indica il formato esadecimale e "%o" richiede la conversione in ottale. Inoltre, "%s" indica che l'argomento successivo è una stringa che termina con il carattere zero; lo stack contiene l'indirizzo di memoria dove reperirla. Il numero di argomenti che si trova sullo stack dovrebbe corrispondere al numero d'indicazioni di conversioni presente nella stringa di formattazione.

Per esempio, la chiamata C

```
printf("x = %d e y = %d\n", x, y);
```

stampa la stringa in cui le due occorrenze di "%d" sono state sostituite con i valori numerici di *x* e *y*. Anche in questo caso, per ragioni di compatibilità con il C, l'ordine dei push è "y", "x" e infine l'indirizzo della stringa di formattazione. Il motivo di questa convenzione è che la *printf* ha un numero variabile di parametri e quindi impilandoli in ordine inverso si è certi di poter individuare facilmente la stringa, che occupa sempre l'ultimo posto. Se i parametri fossero stati impilati da sinistra a destra la stringa di formattazione si troverebbe in qualche locazione precedente dello stack e la procedura *printf* non saprebbe come rintracciarla.

Per quanto riguardo la SPRINTF, il suo primo argomento è il buffer che riceve la stringa da stampare al posto dello standard output. Gli altri argomenti sono gli stessi di PRINTF. La chiamata SSCANF è il contrario di PRINTF nel senso che il suo primo argomento è una stringa contenente gli interi in notazione decimale, ottale o esadecimale, mentre l'argomento successivo è la stringa di formattazione contenente le indicazioni per le conversioni. Gli altri argomenti sono gli indirizzi delle parole di memoria che devono accogliere le informazioni dopo la conversione. Queste subroutine di sistema sono molto versatili e il Paragrafo C.8 ne presenta vari esempi di utilizzo.

C.4.7 Osservazioni sull'insieme d'istruzioni

La definizione ufficiale dell'8088 comprende un prefisso di **deroga di segmento** (*segment override*) che facilita la possibilità di utilizzare indirizzi effettivi che provengono da segmenti differenti; il primo indirizzo di memoria indicato dopo il prefisso viene risolto usando il registro di segmento corrispondente al segmento indicato. Per esempio, l'istruzione

```
ESEG MOV DX,(BX)
```

calcola per prima cosa l'indirizzo di BX usando il segmento extra, poi trasferisce il suo contenuto in DX. Tuttavia non è possibile derogare all'uso del segmento di stack quando l'indirizzo specificato si basa su SP, né all'uso del segmento extra in caso d'istruzione sulle stringhe che utilizza il registro DI. I registri di segmento SS, DS ed ES possono essere usati nelle istruzioni MOV, ma non è permesso trasferire dati immediati nei registri di segmento né utilizzarli nell'operazione XCHG. La programmazione con le deroghe di segmento e con le modifiche dei segmenti di registro è abbastanza complicata e dovrebbe essere evitata laddove possibile. L'interprete usa segmenti di registro costanti e quindi non incappa in problemi del genere.

La maggior parte dei calcolatori mette a disposizione istruzioni in virgola mobile, disponibili direttamente nel processore, in un coprocessore separato, oppure solo attraverso l'interpretazione via software per mezzo di una specie di trap in virgola mobile. La trattazione di queste funzionalità esula dagli scopi di questa appendice.

C.5 Assemblatore

Conclusa la trattazione dell'architettura dell'8088, ci occupiamo della sua programmazione in linguaggio assemblativo e in particolare degli strumenti che mettiamo a disposizione per il suo apprendimento. Affrontiamo per primo l'assemblatore, poi il tracer e infine forniamo qualche informazione pratica circa il loro utilizzo.

C.5.1 Introduzione

Fin qui abbiamo fatto riferimento alle istruzioni mediante i loro **nomi mnemonici**, cioè quelle abbreviazioni facili da ricordare come ADD e CMP. Lo stesso abbiamo fatto con i registri, indicati per mezzo di nomi simbolici quali AX e BP. Un programma scritto usando nomi simbolici per le istruzioni e per i registri si chiama **programma in linguaggio assemblativo**. L'esecuzione di un tale programma richiede in primo luogo la sua traduzione in stringhe binarie che avviene a cura di un programma, l'**assemblatore** (*assembler*). Il risultato del processo di conversione (l'output dell'assemblatore) si chiama **file oggetto**. Molti programmi effettuano chiamate a subroutine assemblate precedentemente e riposte in librerie. L'esecuzione di tali programmi richiede la combinazione del file oggetto appena assemblato e delle subroutine di libreria (a loro volta file oggetto) in un solo **file binario eseguibile** a opera del **linker** ("redattore di collegamenti"). La traduzione si può dire ultimata solo quando il linker ha terminato di costruire il file binario eseguibile a partire da uno o più file oggetto. A questo punto il sistema operativo può leggere il file binario eseguibile caricato in memoria e gestirne l'esecuzione.

Il primo compito dell'assemblatore è redigere una **tabella dei simboli**, che associa i nomi di variabili e costanti simboliche e le etichette ai numeri binari che rappresentano. Le costanti definite direttamente nel programma possono essere aggiunte alla tabella dei simboli senza ulteriori elaborazioni. D'altro canto, le etichette rappresentano indirizzi il cui valore non è immediatamente deducibile e, per calcolarlo, l'assemblatore scandisce il programma riga per riga in quella che è chiamata la **prima passata**. Durante questa fase, l'assemblatore aggiorna un **contatore di locazioni** indicato con il simbolo "." che si pronuncia **dot** ("punto"). Ogni volta che si incontra un'istruzione o un'allocazione di memoria, il contatore di locazioni viene incrementato della dimensione di memoria necessaria a contenere l'elemento incontrato. Perciò, se le prime due istruzioni hanno dimensioni di 2 e 3 byte, un'etichetta che si antepone alla terza istruzione varrà 5.

Per esempio, se un programma inizia con le istruzioni seguenti

```
MOV AX,6
MOV BX,500
L:
```

allora il valore di *L* sarà 5.

All'inizio della **seconda passata** è noto il valore numerico di ogni simbolo. Dato che i nomi mnemonici delle istruzioni hanno valori numerici costanti, è ora possibile cominciare la **generazione del codice**. Le istruzioni sono lette di nuovo una per volta e il loro equivalente binario viene scritto nel file oggetto. Una volta assemblata l'ultima istruzione del programma, il file oggetto è completo.

C.5.2 as88, un assemblatore basato su ACK

Questo paragrafo descrive i dettagli dell'assemblatore/linker *as88*, disponibile nel CD-ROM allegato e presso il nostro sito web. *as88* è in grado di funzionare in associazione con il tracer già presentato, fa parte dell'Amsterdam Compiler Kit (ACK) ed è stato scritto sul modello degli assemblatori UNIX piuttosto che su quello degli assemblatori per MS-DOS o Windows. Questo assemblatore usa il punto esclamativo (!) per i commenti: il contenuto di una parte di riga che segue un punto esclamativo costituisce un commento e non ha alcun effetto sulla produzione del file oggetto. Eventuali righe vuote sono trattate alla stregua dei commenti.

Per la memorizzazione del codice e dei dati dopo la traduzione l'assemblatore usa tre sezioni diverse. Le sezioni sono in relazione con i segmenti di memoria della macchina. La prima è la **sezione di TESTO** per le istruzioni del processore; il segmento dati ospita la **sezione DATI** per l'inizializzazione di memoria necessaria all'avvio del processo; l'ultima sezione si chiama **BSS** (*Block Started by Symbol*), che appartiene ancora al segmento dati ed è destinata all'allocazione della memoria non inizializzata (cioè inizializzata a 0). Ognuna di queste sezioni ha il proprio contatore di locazioni. Lo scopo delle sezioni è permettere all'assemblatore di generare porzioni d'istruzioni e porzioni di dati in modo indipendente, lasciando al linker il compito di riorganizzarle in modo che le istruzioni si trovino tutte nel segmento di testo e i dati nel segmento dati. Ogni riga del codice assemblativo viene tradotta in una sola sezione, ma le righe di codice e le righe di dati possono trovarsi frammiste le une alle altre. Durante l'esecuzione, la sezione di TESTO si trova allocata nel segmento di testo e le sezioni DATI e BSS sono memorizzate (consecutivamente) nel segmento dati.

Le istruzioni o le parole di dati dei programmi in linguaggio assemblativo possono essere precedute da un'etichetta, che può anche occupare un'intera riga (nel qual caso si intende faccia parte dell'istruzione o della parola dati della riga successiva). Per esempio in

```
    CMP AX,ABC
    JE L
    MOV AX,XYZ
L:
```

l'etichetta *L* si riferisce all'istruzione o alla parola di dati che segue. Ci sono due tipi di etichette. Quelle **globali** sono identificatori alfanumerici seguite dai due punti (:), e quelle **locali** possono trovarsi solo nella sezione di TESTO e sono costituite da un'unica cifra seguita dal carattere due punti (:). Le etichette globali devono essere uniche e non possono corrispondere ad alcuna parola chiave del linguaggio, né a un nome mnemonico di un'istruzione. Le etichette locali possono invece occorrere più volte. L'istruzione

```
JE 2f
```

specifica che il salto `JE` (*Jump Equal*) è in avanti (*f* sta per *forward*) verso la successiva etichetta *2*. Analogamente

```
JNE 4b
```

specifica che il salto `JNE` (*Jump Not Equal*) è all'indietro (*b* sta per *backward*) verso la precedente etichetta *4* più vicina.

L'assemblatore permette l'attribuzione di nomi simbolici alle costanti mediante la sintassi

```
identificatore = espressione
```

dove l'identificatore è una stringa alfanumerica come in

```
BLOCKSIZE = 1024
```

Tutti gli identificatori del linguaggio sono definiti dai loro primi otto caratteri: *BLOCKSIZE* e *BLOCKSIZZ* sono lo stesso simbolo, cioè *BLOCKSIZ*. Le espressioni si costruiscono a partire dalle costanti, dai valori numerici e dagli operatori. Le etichette sono considerate costanti perché il loro valore numerico è stabilito al termine della prima passata.

I valori numerici possono essere **ottali** (cominciano per 0), **decimali** o **esadecimali** (cominciano per 0x o 0X). I numeri esadecimali usano le lettere dalla "a" alla "f" (o dalla "A" alla "F") per rappresentare i valori da 10 a 15. Gli operatori interi sono +, –, *, / e %, rispettivamente per addizione, sottrazione, moltiplicazione, divisione e resto. Gli operatori logici sono &, ^ e ~ per le operazioni bit a bit di AND, OR e NOT. Le espressioni possono contenere parentesi quadre, ma non tonde per non creare confusione con le modalità d'indirizzamento.

L'uso delle etichette nelle espressioni richiede cautela. Le etichette d'istruzioni non possono essere sottratte dalle etichette di dati. La differenza tra etichette confrontabili è un valore numerico, ma né le etichette né le loro differenze possono essere usate come costanti all'interno d'espressioni moltiplicative o logiche. Le espressioni consentite nelle definizioni di costanti possono essere utilizzate anche come costanti nelle istruzioni del processore. Alcuni assemblatori mettono a disposizione il costrutto delle macroistruzioni che raggruppano sotto un unico nome un insieme d'istruzioni, ma *as88* non dispone di questa caratteristica.

Ogni linguaggio assemblativo definisce delle direttive che influiscono sul processo di assemblaggio, ma che non sono tradotte in codice binario e si dicono perciò **pseudoistruzioni**. La Figura C.8 elenca le pseudoistruzioni di *as88*.

Il primo gruppo specifica all'assemblatore la sezione in cui vanno elaborate le istruzioni successive. In genere queste indicazioni si scrivono su di una riga separata e possono essere inserite in qualsiasi punto del codice. Per ragioni di carattere implementativo bisogna usare per prima la sezione di TESTO, poi la sezione di DATI e infine la sezione BSS. Una volta specificati questi riferimenti iniziali, le sezioni possono essere usate in un ordine qualsiasi. Inoltre, la prima riga di una sezione dovrebbe cominciare sempre con un'etichetta globale. Non ci sono altri vincoli sull'ordine d'uso delle sezioni.

Istruzione	Descrizione
.SECT .TEXT	Assembla le linee seguenti nella sezione di TESTO
.SECT .DATA	Assembla le linee seguenti nella sezione DATI
.SECT .BSS	Assembla le linee seguenti nella sezione BSS
.BYTE	Assembla gli argomenti come una sequenza di byte
.WORD	Assembla gli argomenti come una sequenza di parole
.LONG	Assembla gli argomenti come una sequenza di long
.ASCII "str"	Salva la stringa str in ASCII senza farla terminare per zero
.ASCIZ "str"	Salva la stringa str in ASCII facendola terminare per zero
.SPACE n	Incrementa il contatore di locazioni di n posizioni
.ALIGN n	Allinea il contatore di locazioni alla locazione che dista al più n byte
.EXTERN	L'identificatore è un nome esterno

Figura C.8 Le pseudoistruzioni di *as88*.

Il secondo gruppo di pseudoistruzioni contiene le dichiarazioni dei tipi di dati per il segmento dati: `.BYTE`, `.WORD`, `.LONG` e stringa. La parola chiave dei primi tre tipi può essere preceduta da un'etichetta facoltativa, seguita poi da una lista d'espressioni costanti separate da virgole. Le stringhe mettono a disposizione due parole chiave, `ASCII` e `ASCIZ`, la cui unica differenza è che la seconda aggiunge in fondo alla stringa un byte di valore zero. Entrambe sono seguite da una stringa tra virgolette. La definizione delle stringhe consente l'uso di molte sequenze di *escape* tra cui menzioniamo quelle elencate nella Figura C.9. Oltre a ciò, è possibile inserire qualsiasi carattere con un carattere barra seguito dalla sua rappresentazione ottale, come \377 (si usano al massimo tre cifre e in questo caso non c'è bisogno di anteporre lo 0).

La pseudoistruzione `SPACE` causa l'incremento del puntatore alle locazioni del numero di byte specificato come argomento. Viene usata spesso dopo un'etichetta del segmento BSS per riservare lo spazio per l'allocazione di una variabile. La parola chiave `ALIGN` serve a far avanzare il puntatore in corrispondenza della prima locazione di memoria che dista 2, 4 o 8 byte dalla locazione corrente per facilitare l'assemblaggio di parole, long, e altro, allocando una quantità di memoria adeguata. Infine, la parola chiave `EXTERN` serve a dichiarare routine o locazioni di memoria come esterne, perché il linker le possa utilizzare per riferimenti esterni. Non è necessario che la loro definizione si trovi nel file corrente; può trovarsi ovunque, basta che sia raggiungibile dal linker.

Benché l'assemblatore sia di per sé uno strumento abbastanza generale, bisogna segnalare alcuni aspetti da tener in conto quando lo si usa con il tracer. L'assemblatore riconosce le parole chiave scritte sia in minuscolo sia in maiuscolo, ma il tracer le visualizza sempre in maiuscolo. L'assemblatore accetta i caratteri "\r" e "\n" come caratteri di invio a capo (fine riga), ma il tracer usa sempre il secondo. Inoltre, anche se l'assemblatore può gestire programmi suddivisi in più file, il tracer richiede invece che il programma sia contenuto in un unico file con estensione ".$". Al suo interno è possibile specificare direttive per includere altri file con il comando

```
#include filename
```


Sequenza di escape	Descrizione
\n	Invio a capo (line feed)
\t	Tab
\\	Backslash
\b	Backspace
\f	Avanzamento modulo (form feed)
\r	Ritorno a margine (carriage return)
\"	Virgolette

Figura C.9 Alcune sequenze di *escape* di *as88*.

Così facendo il file richiesto viene ricopiato all'interno del file ".$" a partire dalla posizione del comando d'inclusione. L'assemblatore verifica che il file da includere sia già stato elaborato e ne carica una copia, il che è particolarmente utile nei casi in cui diversi file usino uno stesso file d'intestazione. Il comando *#include* deve trovarsi all'inizio di una riga, senza essere preceduto da spazi, e il percorso del file deve essere scritto tra virgolette.

Se c'è un unico file sorgente, *pr.s*, allora il progetto si chiama *pr*, e il file che combina i diversi sorgenti (in questo caso uno solo) sarà *pr.$*. Se c'è più di un file sorgente, allora il nome del progetto è il nome del primo file e funge anche da nome del file ".$", generato dall'assemblatore concatenando il contenuto dei file sorgenti. Questo comportamento può essere modificato da linea di comando indicando l'opzione "–o nomeprogetto" prima del primo file sorgente, nel qual caso il file dei sorgenti combinati si chiamerà *nomeprogetto.$*.

L'inclusione dei file e la presenza di più file sorgenti comporta anche alcuni svantaggi. Innanzitutto è necessario che non ci siano nomi di etichette, di variabili e di costanti presenti in file diversi. Inoltre, dato che il file elaborato dall'assemblatore è in definitiva *nomeprogetto.$*, tutti gli eventuali avvisi (*warning*) e messaggi d'errore si riferiscono alle sue righe di codice e non ai singoli file sorgenti. Per progetti molto piccoli, è spesso preferibile inserire tutto il programma in un unico file ed evitare di usare *#include*.

C.5.3 Differenze tra assemblatori dell'8088

L'assemblatore *as88* è modellato sugli assemblatori standard di UNIX e, come tale, differisce dagli altri assemblatori per l'8088: MASM (di Microsoft) e TASM (di Borland), progettati per MS-DOS (e alcuni aspetti di ogni assemblatore sono intimamente legati al sistema operativo). MASM e TASM supportano entrambi tutti i modelli di memoria dell'8088 consentiti in MS-DOS. Per esempio supportano il modello **minuscolo** (*tiny*) di memoria, secondo cui tutto il codice e i tutti dati devono essere contenuti in 64 KB, il modello **piccolo** (*small*), per cui il segmento di codice e il segmento dati possono raggiungere ciascuno i 64 KB, e i modelli **grandi** (*large*), che ammettono molteplici segmenti per il codice e per i dati. La differenza tra questi modelli dipende dall'uso che si fa dei registri di segmento. Il modello grande consente le chiamate a distanza e le modifiche del registro `DS`. Lo stesso processore introduce alcune limitazioni sui registri di segmento (per esempio il registro `CS` non può essere la destinazione di un'istruzione `MOV`). Per semplifi-

care l'attività del tracer, *as88* usa un modello di memoria che somiglia al modello piccolo, anche se, quando è usato senza il tracer, può gestire i registri di segmento senza ulteriori restrizioni.

Questi altri assemblatori non hanno una sezione `.BSS` e inizializzano la memoria solo all'interno delle sezioni DATI. Il file in ingresso all'assemblatore comincia in genere con delle informazioni d'intestazione seguite dalla sezione DATI, indicata con la parola chiave `.data`, seguita a sua volta dal testo del programma dopo la parola chiave `.code`. L'intestazione usa la parola chiave `title` per specificare il nome del programma, la parola chiave `.model` per indicare il modello di memoria e la parola chiave `.stack` per allocare la memoria del segmento di stack. Se il file binario ha estensione *.com*, allora si usa il modello minuscolo di memoria, si considerano i registri come fossero tutti uguali e si riservano i primi 256 byte del segmento condiviso come "Prefisso del Segmento di Programma".

Invece delle direttive `.WORD`, `.BYTE` e `ASCIZ`, questi assemblatori usano la parola chiave `DW` per le parole e `DB` per i byte. Dopo la direttiva `DB` è possibile definire una stringa racchiudendola tra virgolette. Le etichette per la dichiarazione di dati non sono seguite dai due punti. La parola chiave `DUP` serve a inizializzare grandi porzioni di memoria; è preceduta da un numero ed è seguita da un'inizializzazione, come nell'istruzione

```
LABEL DB 1000 DUP (0)
```

che inizializza i 1000 byte di memoria che seguono l'etichetta *LABEL* assegnando loro il byte ASCII zero.

Anche le etichette per le subroutine non sono seguite dai due punti, ma dalla parola chiave `PROC`. L'etichetta è ripetuta anche alla fine della subroutine ed è seguita dalla parola chiave `ENDP`, così l'assemblatore può dedurre l'estensione della subroutine. Non sono supportate le etichette locali.

Le parole chiave delle istruzioni sono identiche per i tre assemblatori MASM, TASM e *as88*. La sorgente segue sempre la destinazione in tutte le istruzioni a due operandi. È pratica comune l'utilizzo dei registri per il passaggio di argomenti alle funzioni invece di impilarli sullo stack. Tuttavia, se le routine assemblate vengono eseguite all'interno di programmi C o C++, è consigliabile usare lo stack per conformarsi al meccanismo di chiamata delle subroutine C. Questa non è una reale differenza, perché anche in *as88* è possibile usare i registri invece dello stack per il passaggio degli argomenti.

La differenza maggiore tra questi tre assemblatori sta nelle chiamate di sistema. In MASM e in TASM, queste vengono effettuate tramite un interrupt di sistema `INT`. L'interrupt più comune è `INT 21H` che serve alla chiamata di funzioni MS-DOS. Il numero di chiamata è inserito in `AX` e ancora una volta si usa un registro per il passaggio degli argomenti. Esistono vettori di interrupt e numeri di interrupt diversi per i diversi dispositivi; per esempio `INT 16H` serve a chiamare le funzioni del BIOS per la tastiera, mentre `INT 10H` riguarda lo schermo. La programmazione di queste funzioni richiede al programmatore la conoscenza di un gran numero d'informazioni che variano da dispositivo a dispositivo. Le chiamate di sistema di UNIX messe a disposizione da *as88* sono invece molto più facili da usare.

C.6 Tracer

Il tracer-debugger è stato pensato per l'esecuzione su un normale terminale con schermo a 24 × 80 caratteri (VT100) e comandi ANSI standard per i terminali. Se lo si vuole eseguire su macchine UNIX o Linux allora si può usare l'emulatore di terminale messo a disposizione dal sistema X-window. Sulle macchine Windows bisogna caricare il driver *ansi.sys* all'interno dei file d'inizializzazione del sistema nel modo che descriviamo in seguito. La Figura C.10 mostra la suddivisione della schermata del tracer in sette finestre.

La finestra in alto a sinistra è la finestra del processore; mostra il contenuto dei registri d'uso generale in notazione decimale, mentre gli altri sono in notazione esadecimale. Poiché il valore numerico del program counter non è di per sé molto informativo, la posizione del sorgente di programma correntemente in esecuzione viene indicata in base alla precedente etichetta globale. Al di sopra del program counter sono elencati i valori di cinque codici di condizione: il simbolo "v" indica un evento di overflow (flag O), mentre le modalità d'incremento o di decremento sono indicate con i simboli ">" e "<" (flag D). Il flag di segno può valere "n" o "p" a seconda che il risultato dell'istruzione sia negativo oppure no. Il flag di zero vale "z" se asserito, mentre il bit di riporto vale "c" quando asserito. Il simbolo "–" indica un flag azzerato.

La finestra in alto al centro visualizza lo stack, rappresentato in esadecimale. La posizione del puntatore allo stack è indicata dalla freccia "=>". L'indirizzo di ritorno delle subroutine è indicato mediante una cifra anteposta al valore esadecimale. La finestra in alto a destra mostra una parte del file sorgente che contiene l'istruzione dell'esecuzione successiva. La freccia "=>" in questa finestra indica la posizione del program counter.

La finestra sotto quella del processore mostra le posizioni del codice contenenti le chiamate di subroutine effettuate più di recente. Appena sotto c'è la finestra dei comandi del tracer in cui viene visualizzata la storia dei comandi impartiti precedentemente, seguita dal cursore. Si noti che ogni comando deve essere seguito dalla pressione del tasto Invio.

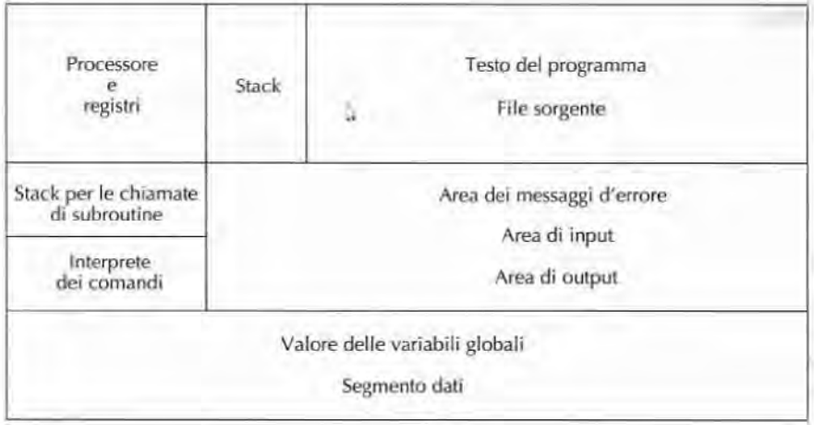


Figura C.10 Finestre del tracer.

La finestra in basso può contenere sei elementi della memoria dei dati globali. Ogni elemento comincia specificando una posizione relativa a un'etichetta, seguita dalla posizione assoluta nel segmento dati, da due punti e quindi da otto byte in esadecimale. Le 11 posizioni successive sono riservate a caratteri, seguiti dalla rappresentazione decimale di quattro parole. I byte, i caratteri e le parole rappresentano tutti lo stesso contenuto di memoria, anche se i caratteri contengono tre byte in più perché all'inizio non è noto se i dati saranno usati come interi con o senza segno, o come stringa.

La prima riga della finestra per l'input/output è riservata all'output per i messaggi d'errore del tracer, la seconda all'input e le righe successive all'output. L'output per gli errori è indicato con la lettera "E", l'input con la lettera "I" e lo standard output con il simbolo ">". La riga di input contiene una freccia "->" a indicare il puntatore alla posizione di prossima lettura. Se il programma chiama *read* o *getchar*, l'input va immesso nell'area di input e bisogna farlo seguire dalla pressione del tasto Invio. La freccia "->" indica la parte della riga che non è stata ancora elaborata.

In genere il tracer usa lo standard input per ricevere comandi e dati, ma è possibile anche preparare un file con i comandi per il tracer e uno con le righe di input che si vogliono far leggere prima che il controllo torni allo standard input. I file per i comandi al tracer hanno estensione *.t*, e quelli per l'input di dati hanno estensione *.i*. Il linguaggio assemblativo ammette l'uso indistinto di caratteri minuscoli o maiuscoli per la scrittura delle parole chiave, delle subroutine di sistema e delle pseudoistruzioni. Durante il processo di assemblaggio viene creato il file d'estensione *.$* e, al proprio interno, le parole chiave sono convertite in maiuscolo e vengono eliminati i caratteri di invio a capo. Dunque, per un dato progetto *pr* possono esistere sei file diversi:

1. *pr.s* per il codice del sorgente assemblativo
2. *pr.$* per il file dei sorgenti combinati
3. *pr.88* per il file di caricamento
4. *pr.i* per lo standard input preparato anticipatamente
5. *pr.t* per i comandi all'interprete preparati anticipatamente
6. *pr.#* per collegare il codice assemblativo al file di caricamento.

L'ultimo file viene usato dal tracer per riempire la finestra in alto a destra e visualizzare il program counter. Infine, il tracer controlla che il file di caricamento sia stato creato successivamente all'ultima modifica apportata al sorgente di programma, altrimenti emette un avviso al riguardo.

C.6.1 Comandi del tracer

La Figura C.11 elenca i comandi del tracer. Il più importante è indicato nella prima riga ed è il comando che si immette con la semplice pressione del tasto Invio per richiedere l'esecuzione di una sola istruzione da parte del processore. Altrettanto importante è il comando *q* di uscita (*quit*), indicato all'ultima riga della tabella. Se si immette come comando un numero, questo provoca l'esecuzione di un numero corrispondente d'istruzioni: il numero *k* equivale a premere il tasto Invio *k* volte. Si ottiene lo stesso effetto se il numero è seguito da un punto esclamativo, *!*, o da una *X*.

Indirizzo	Comando	Esempio	Descrizione
			Esegue una istruzione
#	, ! , X	24	Esegue # istruzioni
/T+#	g , l ,	/start+5g	Continua fino alla riga # dopo etichetta T
/T+#	b	/start+5b	Inserisce breakpoint alla riga # dopo etichetta T
/T+#	c	/start+5c	Rimuove breakpoint alla riga # dopo etichetta T
#	g	108g	Esegue il programma fino alla riga #
	g	g	Esegue il programma finché non torna alla riga corrente
	b	b	Inserisce breakpoint alla riga corrente
	c	c	Rimuove breakpoint dalla riga corrente
	n	n	Esegue programma fino alla riga successiva
	r	r	Esegue programma fino a un breakpoint o alla fine
	=	=	Esegue programma fino allo stesso livello di subroutine
	-	-	Esegue programma fino al livello di subroutine corrente meno 1
	+	+	Esegue programma fino al livello di subroutine corrente più 1
/D+#		/buf+6	Mostra il segmento dati alla posizione etichetta+#
/D+#	d , !	/buf+6d	Mostra il segmento dati alla posizione etichetta+#
	R , CTRL L	R	Aggiorna le finestre
	q	q	Interrompe il tracing, restituisce il controllo all'interprete di comandi

Figura C.11 Comandi del tracer. Ogni comando deve essere seguito da un invio a capo (tasto Enter o Invio). Una casella vuota indica la necessità di digitare il solo carattere di a capo. I comandi che non devono specificare alcun indirizzo hanno la casella della colonna *Indirizzi* vuota. Il simbolo # indica un offset intero.

Il comando *g* può essere usato per spostarsi a una certa riga del file sorgente. In particolare, se preceduto da un numero di riga, il tracer esegue tutte le istruzioni che la precedono; se si usa l'etichetta */T*, con o senza *+#*, il numero di riga dove fermarsi, si calcola a partire dall'etichetta d'istruzione *T*; l'immissione di *g* senza alcun'altra indicazione fa sì che il tracer esegua altri comandi finché non raggiunga di nuovo il numero di riga corrente.

Il comando */label* cambia significato a seconda che lo si usi per etichette di dati o d'istruzioni. Nel primo caso viene aggiunta o sostituita una riga della finestra più in basso con l'insieme di dati che si trova presso quell'etichetta. Nel secondo caso, il comando si comporta come il comando *g*. L'etichetta può essere seguita da un segno + o da un numero (indicato dal simbolo # nella Figura C.11) per specificare un offset relativo all'etichetta.

È possibile impostare un **breakpoint** ("punto d'interruzione") in corrispondenza di un'istruzione tramite il comando *b*, che può essere preceduto da un'etichetta d'istruzione ed eventualmente da un offset. Se durante l'esecuzione si raggiunge un'istruzione con breakpoint, il tracer si ferma; per farlo ripartire basta premere il tasto Invio o un comando d'esecuzione. Se non vengono specificati alcun numero e alcuna etichetta, il breakpoint è associato alla riga corrente. Per rimuovere un breakpoint si usa il comando *c*, preceduto eventualmente dagli stessi parametri opzionali di *b* (etichetta e numero). Esiste il comando d'esecuzione r

che ordina al tracer di continuare l'esecuzione finché non incontri un breakpoint, una chiamata di uscita o la fine del programma.

Il tracer tiene anche traccia del livello di subroutine in cui sta girando il programma. Si può desumere dalla finestra sotto la finestra del processore o anche dai numeri indicati nella finestra dello stack. Ci sono tre comandi che riguardano il livello di subroutine in esecuzione. Il comando – richiede al tracer di proseguire nell'esecuzione finché non raggiunga un livello di subroutine inferiore rispetto a quello attuale. L'effetto di questo comando è l'esecuzione delle istruzioni che mancano al completamento della subroutine correntemente in esecuzione. Al contrario il comando + provoca il proseguimento dell'esecuzione fino al livello di subroutine appena superiore. Il comando = fa proseguire l'esecuzione finché non si incontra lo stesso livello di quello corrente e può essere usato per eseguire una sola subroutine quando ci si trova all'istruzione `CALL`. Quando si usa = la finestra del tracer non mostra i dettagli della subroutine. Si tratta di un comando particolarmente utile in presenza d'istruzioni `LOOP`; infatti l'esecuzione si ferma esattamente alla fine di ogni ciclo.

C.7 Installazione

In questo paragrafo spieghiamo come usare gli strumenti software disponibili. Per prima cosa è necessario individuare il software corrispondente alla propria piattaforma hardware (qui mettiamo a disposizione versioni già compilate per Solaris, UNIX, Linux e Windows). Il software è reperibile nel CD-ROM allegato. Le directory principali del CD-ROM sono *Bigendnx*, *LtlendNx* e *MSWindos*, e ciascuna contiene una directory *assembler* con il materiale. Le tre cartelle radice sono destinate rispettivamente ai sistemi UNIX big endian (per esempio le workstation di Sun), ai sistemi UNIX little endian (per esempio Linux per PC) e ai sistemi Windows.

Dopo averla decompressa o copiata, la directory *assembler* dovrebbe contenere i seguenti file e sottodirectory: *READ_ME*, *bin*, *as_src*, *trce_src*, *examples* e *exercise*. I sorgenti precompilati si trovano nella directory *bin*, ma sono presenti anche nella directory *examples* per comodità.

Per farsi un'idea del funzionamento del sistema, basta portarsi nella directory *examples* ed eseguire

```
t88 HlloWrld
```

Questo comando corrisponde al primo esempio del Paragrafo C.8.

La directory *as_src* contiene il codice sorgente dell'assemblatore. I file sorgenti sono scritti in C e per ricompilarli si usa il comando *make*. Nella directory dei sorgenti è presente anche il file *Makefile* che serve per la compilazione sulle piattaforme POSIX compatibili. Sulle piattaforme Windows la compilazione è svolta dal file batch *make.bat*. Dopo la compilazione potrebbe essere necessario copiare i file eseguibili in un'altra directory, o modificare la variabile d'ambiente PATH, affinché l'assemblatore *as88* e il tracer *t88* siano visibili dalle directory contenenti i sorgenti in linguaggio assemblativo. In alternativa, invece di digitare solo *t88* è possibile specificare l'intero percorso dell'eseguibile.

Sui sistemi Windows 2000 e XP è necessario installare il driver di terminale *ansi.sys* inserendo la riga

```
device=%systemRoot%\System32\ansi.sys
```

nel file di configurazione *config.nt*, che si trova al percorso

```
\winnt\system32\config.nt     (Windows 2000)
\windows\system32\config.nt   (Windows XP)
```

Sui sistemi Windows 95/98/ME il driver va inserito nel file *config.sys*. Nei sistemi UNIX e Linux il driver è in genere già installato.

C.8 Esempi

Nei paragrafi da C.2 a C.4 abbiamo presentato il processore 8088, la sua memoria e le sue istruzioni. Il Paragrafo C.5 ha introdotto il linguaggio assemblativo di *as88* che usiamo in questo testo. Nei Paragrafi C.6 e C.7 abbiamo studiato il tracer e descritto l'installazione degli strumenti software. In teoria queste informazioni dovrebbero già essere sufficienti per la scrittura e il debug di programmi assemblativi con gli strumenti forniti. Ciononostante, a molti lettori potrebbe giovare la presentazione dettagliata di alcuni esempi di programmi assemblativi e il loro debug attraverso il tracer. Perciò qui presentiamo i programmi d'esempio che abbiamo inserito nella directory *examples* del materiale di supporto. Suggeriamo al lettore di assemblarli tutti e di visualizzare la loro esecuzione con il tracer.

C.8.1 Esempio Hello World

Cominciamo con l'esempio *HlloWrld.s* della Figura C.12. Il programma è mostrato nella finestra di sinistra e i numeri di riga sono indicati dopo il punto esclamativo (!), il simbolo usato dall'assemblatore per i commenti. Le prime tre righe contengono le definizioni di costanti per associare la rappresentazione interna di due chiamate di sistema e di un file di output ai loro nomi convenzionali.

La pseudoistruzione `.SECT` della riga 4 specifica che le righe successive fanno parte della sezione di TESTO, ovvero che si tratta d'istruzioni del processore. Analogamente, la riga 17 indica che le righe sottostanti costituiscono dati. La riga 19 inizializza una stringa di dati di 12 byte, compresi lo spazio e il carattere di fine riga (\n).

Le righe 5, 18 e 20 contengono etichette, individuabili dalla presenza dei due punti. Queste etichette rappresentano valori numerici e in questo sono simili a costanti; tuttavia, il valore numerico non è dato, ma deve essere calcolato dall'assemblatore. L'etichetta *start* si trova all'inizio della sezione di TESTO e quindi vale 0, ma il valore di un'eventuale etichetta successiva nella sezione di TESTO (nell'esempio non ce ne sono) dipenderebbe dal numero di byte che la precedono. Si consideri ora la riga 6 che termina con la differenza di due etichette, cioè con un valore costante. In effetti la riga è equivalente all'istruzione

```
MOV CX,12
```

```
   _EXIT = 1                 ! 1
   _WRITE = 4                ! 2
   _STDOUT = 1               ! 3
.SECT .TEXT                  ! 4
start:                       ! 5
   MOV  CX,de-hw             ! 6
   PUSH CX                   ! 7
   PUSH hw                   ! 8
   PUSH _STDOUT              ! 9
   PUSH _WRITE               !10
   SYS                       !11
   ADD  SP, 8                !12
   SUB  CX,AX                !13
   PUSH CX                   !14
   PUSH _EXIT                !15
   SYS                       !16
.SECT .DATA                  !17
hw:                          !18
.ASCII "Hello World\n"       !19
de: .BYTE 0                  !20
```

(a)

```
CS:  00    DS=SS=ES:  002          MOV  CX,de-hw   ! 6
AH:00  AL:0c    AX:    12          PUSH CX         ! 7
BH:00  BL:00    BX:     0          PUSH HW         ! 8
CH:00  CL:0c    CX:    12          PUSH _STDOUT    ! 9
DH:00  DL:00    DX:     0          PUSH _WRITE     !10
SP:  7fd8  SF  O D S Z C  =>0004   SYS             !11
BP:  0000  CC   - > p - -   0001 =>ADD  SP,8       !12
SI:  0000    IP:000c:PC     0000   SUB  CX,AX      !13
DI:  0000    start + 7      000c   PUSH CX         !14
                                E
                                l
hw
■                               > Hello  World\n
hw + 0 = 0000:  48 65 6c 6c 6f 20 57 6f   Hello  World  25928
```

(b)

Figura C.12 (a) Programma *HlloWrld.s*. (b) Finestre del tracer durante la sua esecuzione.

con la differenza che, nel primo caso, il calcolo della lunghezza della stringa è lasciato all'assemblatore e non al programmatore. Il valore indicato dalla differenza corrisponde alla quantità di spazio riservato alla stringa della riga 19 nella sezione dati. L'istruzione `MOV` della riga 6 richiede la copia di *de–hw* in `CX`.

Le righe da 7 a 11 mostrano il funzionamento dell'invocazione di una chiamata di sistema. Le cinque righe sono la traduzione in codice assemblativo della chiamata di funzione C

```
write(1, hw, 12);
```

il cui primo parametro è il descrittore di file per lo standard output (1), il secondo è l'indirizzo della stringa da stampare (*hw*) e il terzo è la lunghezza della stringa (12). Le righe dalla 7 alla 9 impilano questi parametri sullo stack in ordine inverso, secondo la convenzione del C e quella adottata dal tracer. La riga 10 impila sullo stack il numero della chiamata di sistema `write` (4) e la 11 effettua la chiamata vera e propria. Questa sequenza di chiamata ricalca il funzionamento dei programmi in linguaggio assemblativo scritti per i sistemi UNIX (o Linux), ma per eseguirla direttamente su altri sistemi operativi dovrebbe essere leggermente modificata per aderire alle convenzioni di quei sistemi. Tuttavia, l'assemblatore *as88* e il tracer *t88* usano le convenzioni di chiamata di UNIX anche se sono eseguiti su sistemi Windows.

La chiamata di sistema della riga 11 esegue la scrittura vera e propria. La riga 12 ripristina lo stack alla posizione precedente alla chiamata, spostando il puntatore allo stack indietro di quattro parole di 2 byte ciascuna. Se l'operazione di scrittura ha successo, il numero di byte scritti viene restituito nel registro `AX`. La riga 13 sottrae il risultato della chiamata di sistema della riga 11 al valore del registro `CX`, contenente la lunghezza della stringa originale, per stabilire se la chiamata ha avuto successo, ossia per stabilire se sono stati scritti tutti i byte. Di conseguenza, lo stato di uscita del programma sarà 0 soltanto in caso di successo. Le righe 14 e 15 preparano l'esecuzione della chiamata di sistema `exit` (riga 16) impilando sullo stack lo stato d'uscita e il codice della funzione `EXIT`.

Le istruzioni `MOV` e `SUB` usano il primo operando come destinazione e il secondo come sorgente. Questa è la convenzione adottata dal nostro assemblatore, altri potrebbero usare l'ordine inverso; non c'è alcuna ragione particolare per preferire un ordinamento a un altro.

Proviamo ad assemblare e a eseguire *HlloWrld.s*. Forniamo le indicazioni sui procedimenti da seguire sulle piattaforme UNIX e Windows; le indicazioni per UNIX dovrebbero valere anche per Linux, Solaris, MacOS X e per altre varianti di UNIX. Si comincia con l'aprire una finestra dell'interprete di comandi (la shell). Su Windows bisogna cliccare in sequenza su

```
Start > Programmi > Accessori > Prompt dei comandi
```

Quindi ci si porta nella directory *examples* con il comando *cd* (*Change Directory*). L'argomento di questo comando dipende dalla locazione del sistema in cui è stato copiato il materiale di supporto. Una volta entrati nella directory si può verificare la presenza dei file binari dell'assemblatore e del tracer con il comando UNIX *ls* o con il comando Windows *dir*. I file binari si chiamano rispettivamente *as88* e *t88*; sui sistemi Windows hanno estensione *.exe*, ma non c'è bisogno di specificare l'estensione per lanciare la loro esecuzione. Se i file suddetti non si trovano nella directory, bisogna trovarli e copiarli al suo interno.

È ora possibile passare all'assemblaggio digitando il comando

```
as88 HlloWrld.s
```

Se il comando produce un errore nonostante l'assemblatore sia presente nella cartella, si può provare a digitare

```
./as88 HlloWrld.s
```

nei sistemi UNIX o

```
.\as88 HlloWrld.s
```

nei sistemi Windows.

Se l'assemblaggio termina correttamente, verranno visualizzati i messaggi:

```
Project HlloWrld listfile HlloWrld.$
Project HlloWrld num file HlloWrld.#
Project HlloWrld loadfile HlloWrld.88.
```

che indicano la creazione dei file corrispondenti. Se non ci sono messaggi d'errore, si procede con l'invocazione del tracer:

```
t88 HlloWrld
```

che ne apre le finestre. La freccia nel riquadro in alto a destra punta ora sull'istruzione

```
MOV CX,de-hw
```

della riga 6. A questo punto si può digitare un "a capo" (il tasto Enter o Invio sulle tastiere dei PC) che provoca l'esecuzione di un'istruzione, per cui l'istruzione ora puntata è

```
PUSH CX
```

e CX contiene adesso il valore 12 (nella finestra di sinistra). Dopo un altro "a capo" la finestra centrale contiene il valore 000c, cioè 12 in esadecimale. Questa finestra mostra lo stack, che adesso contiene una parola di valore 12. Se si preme Invio ancora tre volte si eseguono le tre PUSH delle righe 8, 9 e 10, dopodiché lo stack conterrà quattro elementi e il program counter varrà 000b (indicato nella finestra di sinistra).

Dopo un altro "a capo" viene eseguita la chiamata di sistema che produce la stampa della stringa "Hello World\n" nella finestra in basso a destra. Adesso SP vale 0x7ff0, ma dopo un altro "a capo" viene incrementato di 8 e diventa 0x7ff8. Dopo altri quattro invii a capo, la chiamata di sistema viene completata e l'esecuzione del tracer si interrompe.

Per comprendere a fondo il funzionamento dell'assemblaggio può essere utile aprire il file *hlloWrld.s* con un editor. A tal fine conviene evitare di usare programmi come *Word* perché potrebbero formattare il sorgente in modo scorretto. Sulle macchine UNIX si possono usare per esempio *ex*, *vi*, o *emacs*; sui sistemi Windows si può aprire *Notepad*, un editor molto semplice reperibile cliccando in sequenza su

```
Start > Programmi > Accessori > Notepad
```

Consigliamo di modificare la stringa alla riga 19 per visualizzare un messaggio diverso. Dopo aver salvato il file, si può procedere al suo assemblaggio ed eseguirlo con il tracer. Questa semplice modifica è un primo approccio alla programmazione in linguaggio assemblativo.

C.8.2 Esempio con i registri d'uso generale

L'esempio che presentiamo ora mostra più in dettaglio l'uso dei registri ed evidenzia una delle insidie della moltiplicazione nell'8088. La parte sinistra della Figura C.13 mostra una porzione del programma *genReg.s*, mentre la parte destra raffigura la finestra dei registri del tracer in due momenti diversi dell'esecuzione del programma. La Figura C.13(b) mostra lo stato dei registri dopo l'esecuzione della riga 7. L'istruzione alla riga 4

```
MOV AX,258
```

carica il valore 258 in AX, che equivale a caricare 1 in AH e 2 in AL. Alla riga 5 si effettua la somma di AL e AH, dopodiché AH vale 3. Alla riga 6 viene copiato il contenuto della variabile *times* (10) in CX, alla riga 7 viene caricato l'indirizzo della variabile *muldat* (2) in BX; l'indirizzo di *muldat* è 2, perché questa si trova in corrispondenza del secondo byte della sezione DATI. Questa è la situazione fotografata nella Figura C.13(b): AH vale 3, AL vale 2 e AX vale 770, in quanto 3 × 256 + 2 = 770.

L'istruzione successiva (riga 8) copia il contenuto di *muldat* in AX. Dunque, dopo un ulteriore invio a capo, AX conterrà 625.

Siamo ora pronti per entrare in un ciclo che moltiplica il contenuto di AX alla parola che si trova all'indirizzo 2(BX), cioè all'indirizzo *muldat* + 2, che vale 2. La destinazione implicita dell'istruzione di moltiplicazione è la coppia di registri DX:AX. Dopo la prima iterazione del ciclo il risultato può essere contenuto in una sola parola, perciò AX contiene il risultato (1250) e DX resta a 0. La Figura C.13(c) mostra il contenuto dei registri dopo 7 moltiplicazioni successive.

All'inizio del ciclo AX valeva 625, perciò il risultato delle sette moltiplicazioni per 2 è 80.000. Questo numero non può più essere contenuto in AX, bensì occupa ora i 32 bit dei registri DX:AX accoppiati. DX vale 1 e AX vale 14.464. L'istruzione LOOP decrementa il registro CX di un'unità a ogni iterazione. Poiché all'inizio dell'esecuzione CX valeva 10, dopo sette operazioni MUL (e dopo sole sei iterazioni dell'istruzione LOOP) vale 4.


```
start:                  !3
    MOV  AX,258         !4
    ADDB AH,AL          !5
    MOV  CX,(times)     !6
    MOV  BX,muldat      !7
    MOV  AX,(BX)        !8
llp: MUL  2(BX)         !9
    LOOP llp            !10
.SECT .DATA             !11
times: .WORD 10         !12
muldat:.WORD 625,2      !13
```

(a)

```
CS: 00    DS=SS=ES: 002
AH:03 AL:02    AX:   770
BH:00 BL:02    BX:     2
CH:00 CL:0a    CX:    10
DH:00 DL:00    DX:     0
SP:  7fe0  SF  O D S Z C
BP:  0000  CC  - > p - -
SI:  0000  IP:0009:PC
DI:  0000  start + 4
```

(b)

```
CS: 00    DS=SS=ES: 002
AH:38 AL:80    AX: 14464
BH:00 BL:02    BX:     2
CH:00 CL:04    CX:     4
DH:00 DL:01    DX:     1
SP:  7fe0  SF  O D S Z C
BP:  0000  CC  v > p - c
SI:  0000  IP:0011:PC
DI:  0000  start + 7
```

(c)

Figura C.13 (a) Porzione del programma *genReg.s*. (b) Finestra dei registri del tracer dopo l'esecuzione della riga 7, (c) dopo l'esecuzione di sette moltiplicazioni (sei iterazioni del ciclo).

Il problema si verifica al momento della moltiplicazione successiva. La moltiplicazione coinvolge AX ma non DX, perciò MUL moltiplica AX (14.464) per 2 producendo 28.928. Il risultato è inserito in AX e DX viene azzerato, il che è algebricamente errato.

C.8.3 Istruzioni di chiamata e puntatori ai registri

L'esempio successivo, *vecprod.s*, è un breve programma per il calcolo del prodotto scalare di due vettori, *vec1* e *vec2*. È mostrato nella Figura C.14.

La prima parte del programma prepara la chiamata di *vecmul* salvando SP in BP e impilando gli indirizzi di *vec2* e *vec1* sullo stack, di modo che *vecmul* possa accedervi. La riga 8 carica in CX la lunghezza in byte del vettore. La riga 9 fa scorrere questo valore di un bit a destra, così CX contiene ora la lunghezza del vettore in parole. Questo valore viene impilato sullo stack alla riga 10. La riga 11 effettua la chiamata a *vecmul*.

Ancora una volta vale la pena sottolineare come gli argomenti delle subroutine vengano impilati in ordine inverso per rispettare la convenzione vigente per le chiamate del linguaggio C. Così facendo, *vecmul* può essere invocata anche all'interno di un programma C con la chiamata

```
vecmul(count, vec1, vec2)
```

Durante l'esecuzione dell'istruzione CALL, l'indirizzo di ritorno viene impilato sullo stack e, usando il tracer, si evince che questo indirizzo è 0x0011.

La prima istruzione della subroutine è PUSH del puntatore base, BP, alla riga 22. BP viene salvato perché servirà a indirizzare gli argomenti e le variabili locali della subroutine. Successivamente viene copiato il puntatore allo stack (riga 23) di modo che il nuovo valore del puntatore base punti al suo vecchio valore.

A questo punto si è pronti per il caricamento nei registri degli argomenti e per l'allocazione dello spazio per le variabili locali. Le tre righe successive prelevano gli argomenti dallo stack e li salvano nei registri. Si ricordi che lo stack è orientato alla parola, perciò i suoi indirizzi dovrebbero essere sempre pari. L'indirizzo di ritorno si trova appena dopo il vecchio puntatore base, perciò viene indirizzato con 2(BP). Segue l'argomento *count*, indirizzato con

```
_EXIT     = 1                    ! 1   definizione del valore di _EXIT
_PRINTF   = 127                  ! 2   definizione del valore di _PRINTF
.SECT .TEXT                      ! 3   inizio del segmento di TESTO
inpstart:                        ! 4   definizione dell'etichetta inpstart
      MOV  BP,SP                 ! 5   salvataggio di SP in BP
      PUSH vec2                  ! 6   push dell'indirizzo di vec2
      PUSH vec1                  ! 7   push dell'indirizzo di vec1
      MOV CX,vec2-vec1           ! 8   CX = numero di byte nel vettore
      SHR  CX,1                  ! 9   CX = numero di parole nel vettore
      PUSH CX                    ! 10  push del numero di parole (count)
      CALL vecmul                ! 11  chiamata di vecmul
      MOV  (inprod),AX           ! 12  copia di AX
      PUSH AX                    ! 13  push del risultato per la stampa
      PUSH pfmt                  ! 14  push dell'indirizzo della stringa di formattazione
      PUSH _PRINTF               ! 15  push del codice di funzione di PRINTF
      SYS                        ! 16  chiamata della funzione PRINTF
      ADD  SP,12                 ! 17  ripristino dello stack
      PUSH 0                     ! 18  push del codice di stato
      PUSH _EXIT                 ! 19  push del codice di funzione di EXIT
      SYS                        ! 20  chiamata della funzione EXIT

vecmul:                          ! 21  inizio di vecmul(count, vec1, vec2)
      PUSH BP                    ! 22  salvataggio di BP nello stack
      MOV  BP,SP                 ! 23  copia di SP in BP per l'accesso agli argomenti
      MOV  CX,4(BP)              ! 24  trasf. di count in CX per il controllo del ciclo
      MOV  SI,6(BP)              ! 25  SI = vec1
      MOV  DI,8(BP)              ! 26  DI = vec2
      PUSH 0                     ! 27  push di 0 sullo stack

1:    LODS                       ! 28  copia di (SI) in AX
      MUL  (DI)                  ! 29  moltiplicazione di AX per (DI)
      ADD  -2(BP),AX             ! 30  somma di AX con valore accumulato in memoria
      ADD  DI,2                  ! 31  incremento di DI per puntare a elemento successivo
      LOOP 1b                    ! 32  se CX > 0, vai all'etichetta 1b
      POP  AX                    ! 33  pop della cima dello stack in AX
      POP  BP                    ! 34  ripristino di BP
      RET                        ! 35  ritorno da subroutine

.SECT .DATA                      ! 36  inizio del segmento DATI
pfmt:. ASCIZ "Inner product is: %d\n"  ! 37  definizione di stringa
.ALIGN 2                         ! 38  forza allineamento a indirizzo pari
vec1:.WORD 3,4,7,11,3            ! 39  vettore 1
vec2:.WORD 2,6,3,1,0             ! 40  vettore 2
.SECT .BSS                       ! 41  inizio del segmento BSS
inprod:   .SPACE 2               ! 42  allocazione dello spazio per inprod
```

Figura C.14 Programma *vecprod.s*.

4(BP), che viene caricato in CX alla riga 24. Le righe 25 e 26 caricano *vec1* e *vec2* in SI e DI. Questa subroutine utilizza una variabile locale (inizializzata a 0) per salvare il risultato. A tal fine, la riga 27 esegue il push del valore 0.

La Figura C.15 mostra lo stato del processore appena prima del primo ingresso nel ciclo della riga 28. La finestra stretta che si trova in alto e al centro della figura (alla destra dei registri) mostra lo stack. Alla base dello stack c'è l'indirizzo di *vec2* (0x0022), seguito da quello di *vec1* (0x0018) e dal terzo argomento, cioè il numero d'elementi di ciascun vettore (0x0005).

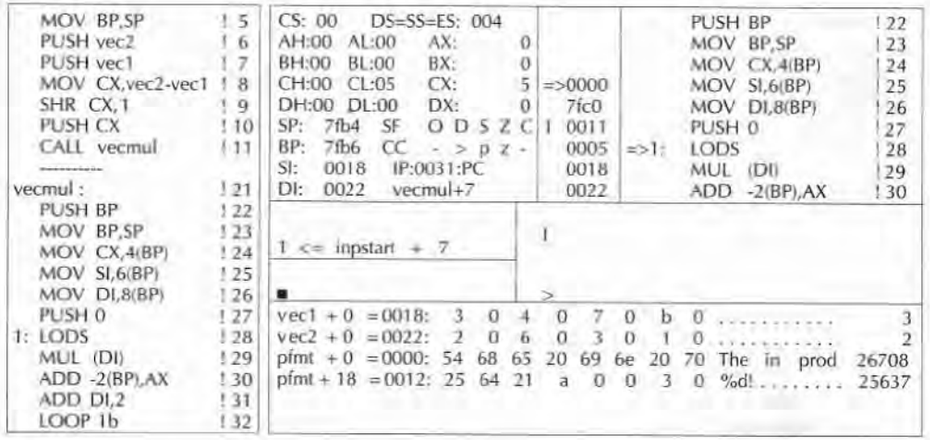


Figura C.15 Esecuzione di *vecprod.s* al primo ingresso nel ciclo della riga 28.

Poi c'è l'indirizzo di ritorno (0x0011); il numero 1 alla sua sinistra indica che si tratta di un indirizzo di ritorno di un livello inferiore rispetto al livello del programma principale. La stessa indicazione è presente anche nella finestra sotto i registri, ma questa volta l'indirizzo è indicato in maniera simbolica. Risalendo lo stack, sopra l'indirizzo di ritorno si trova il vecchio valore di BP (0x7fc0) e poi lo zero impilato alla riga 27. La freccia che punta a questo valore indica il valore puntato da SP. La finestra alla destra dello stack mostra un frammento del testo del programma, dove la freccia indica l'istruzione successiva da eseguire.

Esaminiamo ora il ciclo che comincia alla riga 28. L'istruzione LODS carica in AX una parola di memoria dal segmento dati in modalità registro indiretto (tramite il registro SI). Poiché il flag della modalità d'incremento è asserito, LODS si autoincrementa, perciò dopo l'esecuzione della 28, SI punta al successivo elemento di *vec1*.

Per visualizzare graficamente questo meccanismo, basta avviare il tracer con il comando

```
t88 vecprod
```

e attendere il caricamento del programma, quindi digitare

```
/vecmul+7b
```

(seguito da un invio a capo) per inserire un breakpoint alla riga dell'istruzione LODS. Ricordiamo che ogni comando deve essere seguito da un invio a capo e d'ora in avanti eviteremo di ripeterlo. Digitare quindi il comando

```
g
```

affinché il tracer continui a eseguire le istruzioni finché non incontri il breakpoint; si fermerà alla riga contenente LOADS.

La riga 29 moltiplica AX per l'operando sorgente. La parola di memoria richiesta dall'istruzione MUL è prelevata dal segmento dati attraverso il registro DI usato in modalità re-

gistro indiretto. La destinazione di `MUL` è la combinazione di registri `DX:AX`, che è implicita (cioè non menzionata dall'istruzione).

La riga 30 somma il risultato della moltiplicazione alla variabile locale che si trova all'indirizzo dello stack `-2(BP)`. Dato che `MUL` non incrementa automaticamente il proprio operando, bisogna effettuare l'incremento esplicitamente (riga 31), dopodiché `DI` punterà all'elemento successivo del vettore *vec2*.

L'istruzione `LOOP` conclude questa fase di calcolo: `CX` viene decrementato e, se è ancora positivo, il programma torna all'etichetta locale *1* della riga 28. L'espressione *1b* sta a indicare proprio la precedente etichetta locale *1* più vicina rispetto alla posizione corrente. Alla terminazione del ciclo, la subroutine esegue la pop del valore da restituire in `AX` (riga 33), ripristina `BP` (riga 34) e passa il controllo al programma chiamante (riga 35).

Il programma riprende dalla posizione successiva alla chiamata con l'esecuzione dell'istruzione `MOV` alla riga 12. Questa è la prima di cinque istruzioni finalizzate alla stampa del risultato. La chiamata di sistema `printf` ricalca il comportamento della funzione *printf* della libreria C standard. I tre argomenti sono impilati sullo stack alle righe 13-15; si tratta del valore intero da stampare, dell'indirizzo della stringa di formattazione (*pfmt*) e del codice di funzione di `printf` (127). La stringa di formattazione *pfmt* contiene la stringa *%d* a indicare che la chiamata `printf` ha un argomento di tipo intero che deve essere inserito nella stampa.

La riga 17 ripristina lo stack. Poiché l'esecuzione del programma è cominciata alla riga 5 con il salvataggio del puntatore allo stack nella locazione del puntatore base, sarebbe stato possibile ripristinare lo stack anche con l'istruzione

```
MOV SP,BP
```

Il vantaggio di questa soluzione è che il programmatore non deve tenere il conto delle dimensioni raggiunte dal record d'attivazione. La questione è di poca importanza per quel che riguarda il programma principale, ma nelle subroutine può aiutare a liberare lo stack dai dati inutili, per esempio le variabili locali obsolete.

La subroutine *vecmul* può essere inclusa in altri programmi. Se il file sorgente *vecprod.s* è inserito nella riga dei comandi dopo un altro file sorgente in ingresso all'assemblatore, la subroutine è utilizzabile per moltiplicare due vettori di lunghezza fissa. Si consiglia in tal caso di rimuovere le definizioni delle costanti `_EXIT` e `_PRINTF` per evitare di definirle due volte. Basta includere in un punto qualsiasi il file *syscalnr.h* per non doversi più preoccupare di definire le costanti delle chiamate di sistema.

C.8.4 Debugging di un programma per la stampa di array

I programmi degli esempi precedenti sono molto semplici e svolgono la propria funzione correttamente. Veniamo ora all'uso del tracer come strumento di supporto per il debugging di programmi con errori. Il programma seguente dovrebbe stampare l'array d'interi che comincia all'etichetta *vec1*, tuttavia la sua versione iniziale contiene ben tre errori. Vedremo come usare l'assemblatore e il tracer per correggere questi errori; cominciamo con il descrivere il codice.

Per ragioni di comodità abbiamo creato il file d'intestazione *syscalnr.h* dove abbiamo inserito tutte le definizioni di costanti che identificano le chiamate di sistema, per non doverle ridefinire all'interno di ogni programma che ne ha bisogno. La riga 1 del programma

serve proprio a includere il file d'intestazione. *../syscalnr.h* contiene anche le costanti dei descrittori di file

```
STDIN = 0
STDOUT = 1
STDERR = 2
```

aperti automaticamente all'avvio del processo, così come le etichette d'intestazione per i segmenti del testo e dei dati. Tutte queste definizioni sono molto utilizzate, perciò vale la pena includere *../syscalnr.h* all'inizio di tutti i file sorgente scritti in linguaggio assemblativo. Se il codice sorgente è distribuito tra più file, l'assemblatore provvede a includere il file d'intestazione una sola volta per evitare di ridefinire più volte le stesse costanti.

La Figura C.16 contiene il codice il programma *arrayprt*. Abbiamo omesso il commento delle singole istruzioni, perché dovrebbero risultare ormai familiari, perciò abbiamo utilizzato il formato a due colonne. La riga 4 inserisce l'indirizzo dello stack (ancora vuoto) nel registro puntatore base per consentire il successivo ripristino dello stack alla riga 10; basterà poi copiare il puntatore base nel puntatore allo stack come già descritto nell'esempio precedente. Le righe dalla 5 alla 9 impilano gli argomenti nello stack analogamente a quanto già visto negli esempi precedenti. Le righe dalla 22 alla 25 caricano i registri nella subroutine.

Le righe dalla 27 alla 30 mostrano la stampa di una stringa, quelle dalla 31 alla 34 contengono la chiamata di sistema `printf` per un valore intero. Si noti che l'indirizzo della stringa viene impilato alla riga 27, mentre il valore intero viene copiato nello stack alla riga 33. In entrambi i casi l'indirizzo della stringa di formattazione è il primo argomento della `PRINTF`. Le righe dalla 37 alla 39 mostrano come stampare un carattere usando la chiamata di sistema `putchar`.

```
#include "../syscalnr.h"          ! 1
                                  ! 2
.SECT .TEXT                       ! 3
vecpstrt:                         ! 4
   MOV  BP,SP                     ! 5
   PUSH vec1                      ! 6
   MOV  CX,frmatstr-vec1          ! 7
   SHR  CX                        ! 8
   PUSH CX                        ! 9
   CALL vecprint                  ! 10
   MOV  SP,BP                     ! 11
   PUSH 0                         ! 12
   PUSH _EXIT                     ! 13
   SYS                            ! 14
                                  ! 15
.SECT .DATA                       ! 16
vec1:  .WORD 3,4,7,11,3           ! 17
frmatstr: .ASCIZ "%s"             ! 18
                                  ! 19
frmatkop:                         ! 20
.ASCIZ "L'array contiene "        ! 21
frmatint: .ASCIZ " %d"            ! 22
.SECT .TEXT                       ! 23
vecprint:                         ! 24
   PUSH BP                        ! 25
   MOV  BP,SP                     ! 26
   MOV  CX,4(BP)                  ! 27
   MOV  BX,6(BP)                  ! 28
   MOV  SI,0                      ! 29
   PUSH frmatkop                  ! 30
   PUSH frmatstr                  ! 31
   PUSH _PRINTF                   ! 32
   SYS                            ! 33
   MOV  -4(BP),frmatint           ! 34
1: MOV  DI,(BX)(SI)               ! 35
   MOV  -2(BP),DI                 ! 36
   SYS                            ! 37
   INC  SI                        ! 38
   LOOP 1b                        ! 39
   PUSH '\n'                      ! 40
   PUSH _PUTCHAR                  ! 41
   SYS                            ! 42
   MOV  SP,BP                     ! 43
   RET                            ! 44
```

Figura C.16 Programma *arrayprt* prima del debugging.

Dopo aver assemblato il programma, ne lanciamo l'esecuzione con il comando

```
as88 arrayprt.s
```

e riceviamo un errore di operando alla riga 28 del file *arrayprt.$*. Questo file è prodotto dall'assemblatore combinando il file sorgente ai file inclusi ed è il vero input del processo di assemblaggio. Per individuare il contenuto della riga 28 dobbiamo aprire *arrayprt.$* in lettura, mentre non serve a nulla ispezionare *arrayprt.s*, perché i due file non corrispondono, in quanto il primo include il file d'intestazione alla prima riga. La riga 28 di *arrayprt.$* corrisponde alla riga 7 di *arrayprt.s* dal momento che il file d'intestazione, *../syscalnr.h*, contiene 21 righe.

Un modo semplice per visualizzare la riga 28 di *arrayprt.$* su di una macchina UNIX è lanciare il comando

```
head -28 arrayprt.$
```

che mostra le prime 28 righe del file *.$*. L'ultima riga è quella contenente l'errore. Così facendo (o in modo analogo con l'ausilio di un editor) notiamo che l'errore è alla riga 7 del sorgente originale, cioè in corrispondenza dell'istruzione `SHR`. Se confrontiamo la riga di codice con l'elemento corrispondente della Figura C.4 individuiamo il problema: abbiamo dimenticato d'indicare la lunghezza dello scorrimento. Una volta corretta la riga, diventa

```
SHR CX,1
```

È importante sottolineare che l'errore va corretto nel file sorgente originale *arrayprt.s* e *non* nel sorgente ausiliario *arrayprt.$*, rigenerato automaticamente a ogni chiamata dell'assemblatore.

Il nuovo tentativo di assemblare il codice sorgente dovrebbe andare a buon fine. Si può procedere con il lancio del tracer

```
t88 arrayprt
```

Durante l'esecuzione del tracer notiamo che il risultato del programma non corrisponde al vettore che si trova nel segmento dati. Il vettore contiene: 3, 4, 7, 11, 3 ma i valori visualizzati sono: 3, 1024, ... Chiaramente c'è qualcosa che non va.

Per trovare l'errore facciamo ripartire il tracer, ma questa volta procediamo nella simulazione un'istruzione alla volta, esaminando lo stato della macchina appena prima della stampa del valore errato. Il valore da stampare è memorizzato alle righe 32 e 33; poiché la stampa visualizza un valore errato, è questo il punto dove rivolgere l'attenzione. Durante la seconda iterazione del ciclo, `SI` contiene un valore dispari, mentre dovrebbe contenere un numero pari, dato che indirizza parole, non byte. L'errore si trova alla riga 35: `SI` viene incrementato di 1, laddove andrebbe incrementato di 2. Per correggere il baco bisogna modificare la riga in questo modo

```
ADD SI,2
```

Dopo la correzione la lista dei numeri stampati risulta corretta.

Eppure c'è ancora un errore da scovare. Al termine di *vecprint* il tracer visualizza un errore relativo al puntatore allo stack. Il controllo ovvio da effettuare in questo caso è verificare se il valore impilato sullo stack all'atto della chiamata di *vecprint* è uguale al valore che si trova in cima allo stack al momento della chiamata `RET` della riga 41. Il controllo evidenzia che i due valori sono diversi. La soluzione al problema consiste nel sostituire la riga 40 con le due righe:

```
ADD SP,10
POP BP
```

La prima istruzione rimuove dallo stack le 5 parole impilate durante l'esecuzione di *vecprint*, riportando in cima il valore di `BP` salvato alla riga 22. La pop di questo valore verso `BP` ripristina nel registro il valore che aveva prima della chiamata e fa emergere in cima allo stack l'indirizzo di ritorno corretto. Adesso il programma termina correttamente. Il debugging del codice in linguaggio assemblativo somiglia sicuramente più a un'arte che a una scienza, tuttavia risulta molto facilitato dall'uso del tracer rispetto alla "nuda" esecuzione in hardware.

C.8.5 Manipolazione di stringhe e istruzioni su stringhe

Lo scopo principale di questo paragrafo è presentare le istruzioni iterative sulle stringhe. La Figura C.17 mostra due semplici programmi per la manipolazione di stringhe, *strngcpy.s* e *reverspr.s*, contenuti nella directory *examples*. La Figura C.17(a) contiene una subroutine per la copia di una stringa che si avvale di un'altra subroutine, *stringpr*, che non mostriamo qui, ma che è reperibile nel file *stringpr.s*. Per assemblare un programma che comprende più subroutine contenute in file sorgenti diversi, bisogna elencarli tutti dopo il comando *as88*, a cominciare dal file sorgente contenente il programma principale. Quest'ultimo determina il nome del file eseguibile e del file *.$*. Per esempio, nel caso del programma della Figura C.17(a) bisogna digitare

```
as88 strngcpy.s stringpr.s
```

Il programma della Figura C.17(b) stampa le stringhe in ordine inverso. Descriviamo ora i due programmi.

Per sottolineare il fatto che i numeri di riga sono solo commenti, nella Figura C.17(a) abbiamo numerato le righe a partire dalla prima etichetta, omettendo le righe che la precedevano. Il programma principale occupa le righe dalla 2 alla 8: per prima cosa richiama *strngcpy* con due argomenti, la stringa sorgente *mesg2* e la stringa destinazione *mesg1*, perché copi la sorgente nella destinazione.

Diamo uno sguardo a *strngcpy*, cominciando dalla riga 9. La subroutine si aspetta di trovare in cima allo stack gli indirizzi della destinazione e della sorgente. Nelle righe dalla 10 alla 13 si provvede a salvare i registri nello stack di modo che possano essere ripristinati successivamente con le righe dalla 27 alla 30. Alla riga 14 si effettua la copia di `SP` in `BP` come di consueto; adesso `BP` è pronto per essere usato come puntatore base per l'accesso agli argomenti. Ancora una volta il ripristino dello stack avviene mediante la copia di `BP` in `SP`, alla riga 26.

Il cuore della subroutine è l'istruzione `REP MOVSB` della riga 24. L'istruzione `MOVSB` trasferisce il byte puntato da `SI` nella locazione di memoria puntata da `DI`. `SI` e `DI` vengono poi incrementati di un'unità. L'istruzione `REP` crea un ciclo in cui, a ogni iterazione, si esegue `MOVSB` e si decrementa `CX` di 1 dopo ogni trasferimento di un byte. Il ciclo termina quando `CX` raggiunge il valore 0.

Prima di poter eseguire `REP MOVSB`, bisogna però preparare i registri, come svolto nelle righe dalla 15 alla 22. L'indice sorgente `SI` viene copiato nella riga 21 dall'argomento che

```
.SECT .TEXT                                  ! 1
stcstart:                                    ! 2
  PUSH mesg1                                 ! 3
  PUSH mesg2                                 ! 4
  CALL strngcpy                              ! 5
  ADD SP,4                                   ! 6
  PUSH 0                                     ! 7
  PUSH 1                                     ! 8
  SYS                                        ! 9
strngcpy:                                    ! 10
  PUSH CX                                    ! 11
  PUSH SI                                    ! 12
  PUSH DI                                    ! 13
  PUSH BP                                    ! 14
  MOV BP,SP                                  ! 15
  MOV AX,0                                   ! 16
  MOV DI,10(BP)                              ! 17
  MOV CX,-1                                  ! 18
  REPNZ SCASB                                ! 19
  NEG CX                                     ! 20
  DEC CX                                     ! 21
  MOV SI,10(BP)                              ! 22
  MOV DI,12(BP)                              ! 23
  PUSH DI                                    ! 24
  REP MOVSB                                  ! 25
  CALL stringpr                              ! 26
  MOV SP,BP                                  ! 27
  POP BP                                     ! 28
  POP DI                                     ! 29
  POP SI                                     ! 30
  POP CX                                     ! 31
  RET                                        ! 32
.SECT .DATA                                  ! 33
mesg1: .ASCIZ "Dacci un'occhiata\n"          ! 34
mesg2: .ASCIZ "qrst\n"
.SECT .BSS
```

(a)

```
#include "../syscalnr.h"                     ! 1

start: MOV DI,str                            ! 2
    PUSH AX                                  ! 3
    MOV BP,SP                                ! 4
    PUSH _PUTCHAR                            ! 5
    MOVB AL,'\n '                            ! 6
    MOV CX,-1                                ! 7
    REPNZ SCASB                              ! 8
    NEG CX                                   ! 9
    STD                                      ! 10
    DEC CX                                   ! 11
    SUB DI,2                                 ! 12
MOV SI,DI                                    ! 13
    1:LODSB                                  ! 14
    MOV (BP),AX                              ! 15
    SYS                                      ! 16
    LOOP 1b                                  ! 17
    MOVB (BP),'\n '                          ! 18
    SYS                                      ! 19
    PUSH 0                                   ! 20
    PUSH _EXIT                               ! 21
SYS                                          ! 22
.SECT .DATA                                  ! 23
str: .ASCIZ "reverse\n"                      ! 24
```

(b)

Figura C.17 (a) Copia di una stringa (*strngcpy.s*). (b) Stampa di una stringa in ordine inverso (*reverspr.s*).

si trova sullo stack; l'indice destinazione DI è assegnato alla riga 22. Il calcolo del valore di CX è più difficile. Si tenga presente che il carattere di terminazione di una stringa è il byte zero. L'istruzione MOVSB non modifica il flag di zero, a differenza dell'istruzione SCASB (*SCAn Byte String*) che confronta il valore puntato da DI al contenuto di AL e incrementa DI al volo. Anche SCASB è un'istruzione iterabile come MOVSB. Dunque alla riga 15 viene azzerato AX (e anche AL), alla riga 16 viene prelevato dallo stack il valore cui far puntare DI e alla riga 17 CX viene inizializzato a −1. La riga 18 contiene REPNZ SCASB, che effettua il confronto iterato e asserisce la flag zero quando rileva l'uguaglianza. A ogni iterazione del ciclo CX viene decrementato e il ciclo termina quando il flag di zero viene asserito, dal momento che REPNZ controlla sia il valore di CX, sia il contenuto di quel flag. Il numero d'iterazioni di MOVSB si ottiene sottraendo il valore corrente di CX al suo valore iniziale −1, come effettuato alle righe 19 e 20.

Purtroppo si rendono necessarie ben due istruzioni iterative, ma è il prezzo che bisogna pagare per la scelta progettuale secondo cui le istruzioni di trasferimento non possono modificare i codici di condizione. I registri indice devono essere incrementati durante i cicli, e a tal fine è necessario azzerare il flag della modalità d'incremento.

Le righe 23 e 25 stampano la stringa copiata attraverso la chiamata della subroutine *stringpr*, anch'essa contenuta nella directory *examples*. Il suo comportamento è di facile comprensione, perciò ne omettiamo la trattazione.

Il programma della Figura C.17(b) per la stampa invertita comincia con l'usuale riga d'inclusione delle costanti per le chiamate di sistema. La riga 3 impila sullo stack un dato fittizio, mentre la riga 4 fa sì che BP punti alla cima dello stack. Il programma stampa i caratteri ASCII uno per volta, perciò il valore numerico di _PUTCHAR viene posto in cima allo stack. BP punta al carattere da stampare al momento della chiamata di SYS.

Le righe 2, 6 e 7 preparano i registri DI, AL e CX per l'istruzione SCASB. Il registri per la lunghezza e per l'indice di destinazione sono caricati in modo analogo a quanto visto nella routine per la copia di una stringa, ma il valore di AL è ora il carattere di fine riga e non il valore 0. Così facendo, l'istruzione SCASB confronterà i caratteri della stringa *str* con \n e non con 0, e asserirà il flag zero non appena troverà una corrispondenza.

REP SCASB incrementa il registro DI perciò, dopo aver trovato la corrispondenza, l'indice destinazione punta al carattere zero che segue il carattere di fine riga. La riga 12 decrementa DI di 2 di modo che punti all'ultima lettera dell'ultima parola della stringa. Per produrre il risultato desiderato si può procedere percorrendo la stringa in ordine inverso e stampando un carattere alla volta, perciò il flag della modalità d'incremento viene asserito alla riga 10 in modo da invertire la modalità d'incremento dei registri indice adottata dalle istruzioni sulle stringhe. L'istruzione LODSB alla riga 14 copia il carattere in AL e la riga 15 lo inserisce nella locazione dello stack adiacente a _PUTCHAR, così che venga stampato dall'esecuzione di SYS.

Le istruzioni delle righe 18 e 19 stampano un carattere di fine riga e il programma termina con la chiamata _EXIT nel modo usuale.

Questa versione del programma contiene però un baco che può essere individuato eseguendolo con il tracer un'istruzione alla volta.

Il comando */str* inserisce la stringa *str* nell'area dati del tracer. Il valore numerico dell'indirizzo dei dati è visibile, perciò è possibile monitorare la progressione delle posizioni dei registri indice all'interno dei dati rispetto all'indirizzo della stringa.

Il baco si manifesterà solo dopo aver premuto più volte il tasto Invio. Possiamo usare i comandi del tracer per raggiungere il problema più velocemente. Dopo aver avviato il tracer gli si invia il comando *13* per posizionarsi nel mezzo del ciclo. Il comando *b* inserisce un breakpoint in corrispondenza della riga 15. Dopo due invii a capo vediamo apparire la lettera **e** (ultimo carattere della stringa "reverse") nell'area di output. Il comando *r* ordina al tracer di continuare l'esecuzione fino al prossimo breakpoint o fino alla fine del processo. In questo modo possiamo scorrere le lettere impartendo ogni volta il comando *r* finché non ci avviciniamo al problema. Solo allora riprendiamo l'esecuzione un'istruzione alla volta, fino a raggiungere l'istruzione critica.

In alternativa è possibile inserire il breakpoint a una riga specifica, ma a tal fine bisogna tenere presente che è stato incluso il file *syscalnr* e che i numeri di riga vanno spostati di 21. Di conseguenza il breakpoint alla riga 15 può essere inserito con il comando *36b*; questa non è una scelta molto elegante, è preferibile usare l'etichetta globale *start* della riga

2 e impartire il comando */start + 13b* che inserisce il breakpoint nello stesso punto e permette di trascurare la dimensione del file d'intestazione.

C.8.6 Tabella di salto

Molti linguaggi di programmazione mettono a disposizione le istruzioni *case* o *switch* per selezionare una destinazione di salto tra numerose alternative in base al valore di una certa variabile. A volte queste diramazioni "a più vie" (*multiway branch*) sono necessari anche nei programmi in linguaggio assemblativo. Si pensi per esempio a un insieme di chiamate di sistema invocabili tramite un'unica trap `SYS`. Il programma *jumptbl.s* della Figura C.18 mostra un modo per utilizzare questo costrutto nell'assemblatore dell'8088.

Il programma comincia con la stampa della stringa che si trova all'etichetta *strt* e che invita l'utente a immettere una cifra ottale (righe dalla 4 alla 7), quindi legge un carattere dallo standard input (righe 8 e 9). Se il valore di `AX` è minore di 5 il programma lo interpreta come un marcatore di terminazione di file e salta all'etichetta *8* della riga 22, provocando l'uscita del programma con codice d'uscita 0.

In caso contrario, il carattere inserito in `AL` viene esaminato nel modo seguente: se è minore del carattere 0 viene interpretato come uno spazio bianco e viene ignorato saltando alla riga 13, per ricevere un altro carattere. Gli input maggiori della cifra 9 sono considerati errati e vengono mappati nel carattere ASCII dei due punti (:) che segue il carattere 9 nella sequenza dei codici della tabella ASCII.

La riga 17 copia in `BX` il contenuto di `AX`, che a questo punto contiene un valore compreso tra 0 e i due punti. L'istruzione `AND` della riga 18 maschera tutti i bit di `BX` tranne i quattro meno significativi, producendo un risultato che va da 0 a 10 (poiché la cifra 0 corrisponde al codice ASCII 0x30). Il valore di `BX` viene moltiplicato per 2 alla riga 19 con uno scorrimento a sinistra, dato che ci apprestiamo a indirizzare una tabella di parole e non di byte.

La riga 20 contiene un'istruzione di chiamata. L'indirizzo effettivo, ottenuto sommando `BX` al valore numerico dell'etichetta *tbl*, viene caricato nel program counter.

Il programma sceglie una delle dieci subroutine a seconda del carattere ricevuto in ingresso dallo standard input. Ciascuna di queste subroutine impila sullo stack l'indirizzo di un messaggio e salta alla chiamata della subroutine di sistema `_PRINTF` condivisa da tutte e dieci.

Per capire come funziona il programma bisogna sapere che le istruzioni `JMP` e `CALL` caricano un certo indirizzo del segmento di testo nel `PC`. Questo indirizzo non è altro che un numero, poiché il processo di assemblaggio sostituisce tutti gli indirizzi con i loro valori binari. Alla riga 50 questi valori binari vengono utilizzati per inizializzare un array nel segmento dati. L'array che comincia alla locazione *tbl* contiene gli indirizzi *rout0*, *rout1*, *rout2*, ecc., ciascuno lungo due byte. La necessità di usare indirizzi di due byte giustifica lo scorrimento di un bit che abbiamo incontrato alla riga 19. Un array di questo tipo si dice **tabella di salto** (*dispatch table*).

Osserviamo il funzionamento di queste routine descrivendo la routine *erout* che occupa le righe dalla 43 alla 48 e che gestisce il caso di un carattere esterno all'intervallo delle cifre. Alla riga 43 viene posto in cima allo stack l'indirizzo del messaggio (che si trova in `AX`), poi viene impilato il numero della chiamata di sistema `_PRINTF`, si effettua la chiamata, si ripristina lo stack e infine la routine restituisce il controllo al chiamante. Anche le altre nove routine da *rout0* a *rout8* caricano in `AX` l'indirizzo del loro messaggio privato, poi saltano alla seconda riga di *erout* per stampare il messaggio e terminare l'esecuzione della subroutine.


```
#include "../syscalnr.h"          ! 1
.SECT .TEXT                       ! 2
jumpstrt:                         ! 3
    PUSH strt                     ! 4
    MOV  BP,SP                    ! 5
    PUSH _PRINTF                  ! 6
    SYS                           ! 7
    PUSH _GETCHAR                 ! 8
1:  SYS                           ! 9
    CMP  AX,5                     ! 10
    JL   8f                       ! 11
    CMPB AL,'0'                   ! 12
    JL   1b                       ! 13
    CMPB AL,'9'                   ! 14
    JLE  2f                       ! 15
    MOVB AL,'9'+1                 ! 16
2:  MOV  BX,AX                    ! 17
    AND  BX,0Xf                   ! 18
    SAL  BX,1                     ! 19
    CALL tbl(BX)                  ! 20
    JMP  1b                       ! 21
8:  PUSH 0                        ! 22
    PUSH _EXIT                    ! 23
    SYS                           ! 24

rout0:  MOV  AX,mes0              ! 25
        JMP  9f                   ! 26
rout1:  MOV  AX,mes1              ! 27
        JMP  9f                   ! 28
rout2:  MOV  AX,mes2              ! 29
        JMP  9f                   ! 30
rout3:  MOV  AX,mes3              ! 31
        JMP  9f                   ! 32
rout4:  MOV  AX,mes4              ! 33
        JMP  9f                   ! 34
rout5:  MOV  AX,mes5              ! 35
        JMP  9f                   ! 36
rout6:  MOV  AX,mes6              ! 37
        JMP  9f                   ! 38
rout7:  MOV  AX,mes7              ! 39
        JMP  9f                   ! 40
rout8:  MOV  AX,mes8              ! 41
        JMP  9f                   ! 42
erout:  MOV  AX,emes              ! 43
9:      PUSH AX                   ! 44
        PUSH _PRINTF              ! 45
        SYS                       ! 46
        ADD  SP,4                 ! 47
        RET                       ! 48

.SECT .DATA                                                          ! 49
tbl: .WORD rout0,rout1,rout2,rout3,rout4,rout5,rout6,rout7,rout8,rout8,erout  ! 50
mes0: .ASCIZ "La cifra è uno zero.\n"                                ! 51
mes1: .ASCIZ "La cifra è un uno.\n"                                  ! 52
mes2: .ASCIZ "La cifra è un due.\n"                                  ! 53
mes3: .ASCIZ "La cifra è un tre.\n"                                  ! 54
mes4: .ASCIZ "La cifra è un quattro.\n"                              ! 55
mes5: .ASCIZ "La cifra è un cinque.\n"                               ! 56
mes6: .ASCIZ "La cifra è un sei.\n"                                  ! 57
mes7: .ASCIZ "La cifra è un sette.\n"                                ! 58
mes8: .ASCIZ "La cifra non è una cifra ottale valida.\n"             ! 59
emes: .ASCIZ "Questa non è una cifra ottale.\n"                      ! 60
strt: .ASCIZ "Inserisci una cifra ottale valida seguita da invio. Per uscire inserisci
          un carattere di fine file.\n"                              ! 61
```

Figura C.18 Programma che implementa una diramazione a più vie usando una tabella di salto.

Per abituarsi all'uso delle tabelle di salto, conviene simulare il programma con il tracer e osservarne il comportamento per diversi caratteri di input. Come esercizio, esortiamo il lettore a modificare il programma perché risponda a ogni input con un'azione adeguata: per esempio potrebbe stampare un messaggio d'errore tutte le volte che viene inserito un carattere che non corrisponde a una cifra ottale.

C.8.7 File: accesso diretto e accesso bufferizzato

La Figura C.19 mostra il programma *InFilBuf.s* che illustra l'I/O di file con accesso diretto. Un file è pensato come un insieme di righe che possono avere lunghezze diverse. Il programma comincia con la lettura del file e costruisce una tabella il cui elemento *n* contiene la locazione del file in cui inizia la riga *n*. Se poi giunge una richiesta per una riga del file, si ac-

```
#include "../syscalnr.h"  ! 1
bufsiz = 512              ! 2
.SECT .TEXT               ! 3
infbufst:                 ! 4
    MOV  BP,SP            ! 5
    MOV  DI,linein        ! 6
    PUSH _GETCHAR         ! 7
1:  SYS                   ! 8
    CMPB AL,'\n '         ! 9
    JL   9f               ! 10
    JE   1f               ! 11
    STOSB                 ! 12
    JMP  1b               ! 13
1:  PUSH 0                ! 14
    PUSH linein           ! 15
    PUSH _OPEN            ! 16
    SYS                   ! 17
    CMP AX,0              ! 18
    JL  9f                ! 19
    MOV (fildes),AX       ! 20
    MOV SI,linh+2         ! 21
    MOV BX,0              ! 22
1:  CALL fillbuf          ! 23
    CMP  CX,0             ! 24
    JLE  3f               ! 25
2:  MOVB AL,'\n '         ! 26
    REPNE SCASB           ! 27
    JNE  1b               ! 28
    INC  (count)          ! 29
    MOV  AX,BX            ! 30
    SUB  AX,CX            ! 31
    XCHG SI,DI            ! 32
    STOS                  ! 33
    XCHG SI,DI            ! 34
    CMP  CX,0             ! 35
    JNE  2b               ! 36
    JMP  1b               ! 37
9:  MOV  SP,BP            ! 38
    PUSH linein           ! 39
    PUSH errmess          ! 40
    PUSH _PRINTF          ! 41
    SYS                   ! 42
    PUSH _EXIT            ! 43
    PUSH _EXIT            ! 44
    SYS                   ! 45
3:  CALL getnum           ! 46
    CMP  AX,0             ! 47
    JLE  8f               ! 48
    MOV  BX,(curlin)      ! 49
    CMP  BX,0             ! 50
    JLE  7f               ! 51
    CMP  BX,(count)       ! 52
    JG   7f               ! 53
    SHL  BX,1             ! 54
    MOV  AX,linh-2(BX)    ! 55
    MOV  CX,linh(BX)      ! 56
    PUSH 0                ! 57
    PUSH 0                ! 58
    PUSH AX               ! 59
    PUSH (fildes)         ! 60
    PUSH _LSEEK           ! 61
    SYS                   ! 62
    SUB  CX,AX            ! 63
    PUSH CX               ! 64
    PUSH buf              ! 65
    PUSH (fildes)         ! 66
    PUSH _READ            ! 67
    SYS                   ! 68
    ADD  SP,4             ! 69
    PUSH 1                ! 70
    PUSH _WRITE           ! 71
    SYS                   ! 72
    ADD  SP,14            ! 73
    JMP  3b               ! 74
8:  PUSH scanerr          ! 75
    PUSH _PRINTF          ! 76
    SYS                   ! 77
    ADD  SP,4             ! 78
    JMP  3b               ! 79
7:  PUSH 0                ! 80
    PUSH _EXIT            ! 81
    SYS                   ! 82
fillbuf:                  ! 83
    PUSH bufsiz           ! 84
    PUSH buf              ! 85
    PUSH (fildes)         ! 86
    PUSH _READ            ! 87
    SYS                   ! 88
    ADD  SP,8             ! 89
    MOV  CX,AX            ! 90
    ADD  BX,CX            ! 91
    MOV  DI,buf           ! 92
    RET                   ! 93

getnum:                   ! 94
    MOV  DI,linein        ! 95
    PUSH _GETCHAR         ! 96
1:  SYS                   ! 97
    CMPB AL,'\n '         ! 98
    JL   9b               ! 99
    JE   1f               !100
    STOSB                 !101
    JMP  1b               !102
1:  MOVB (DI),' '         !103
    PUSH curlin           !104
    PUSH numfmt           !105
    PUSH linein           !106
    PUSH _SSCANF          !107
    SYS                   !108
    ADD  SP,10            !109
    RET                   !110

.SECT .DATA               !111
errmess:                  !112
.ASCIZ "Apertura di %s non riuscita\n" !113
numfmt:  .ASCIZ "%d"      !114
scanerr:                  !115
.ASCIZ "Digitare un numero.\n" !116
.ALIGN 2                  !117
.SECT  .BSS               !118
linein:  .SPACE 80        !119
fildes:  .SPACE 2         !120
linh:    .SPACE 8192      !121
curlin:  .SPACE 4         !122
buf:     .SPACE bufsiz+2  !123
count:   .SPACE 2         !124
```

Figura C.19 Programma per l'accesso diretto e per l'accesso bufferizzato di file.

cede all'elemento corrispondente della tabella e si legge la riga con le chiamate di sistema `lseek` e `read`. Il programma accetta come primo dato il nome del file immesso da standard input. Il codice è diviso in porzioni abbastanza indipendenti che possono essere modificate per assolvere ad altre funzioni.

Le prime cinque righe di codice servono a definire i numeri delle chiamate di sistema e la dimensione del buffer, e ad assegnare la cima dello stack al puntatore base, come di consueto. Le righe dalla 6 alla 13 leggono il nome del file dallo standard input e lo memorizzano come stringa all'etichetta *linein*. Se il nome del file non è seguito da un carattere di fine riga, viene visualizzato un messaggio d'errore e il processo termina con un codice di stato diverso da zero (righe da 38 a 45). Si noti che l'indirizzo del nome del file viene impilato alla riga 39, mentre l'indirizzo del messaggio d'errore alla riga 40. Osserviamo che il messaggio d'errore (riga 113) contiene la combinazione di caratteri *%s*, cioè una richiesta per una stringa secondo il formato di `_PRINTF`. La richiesta viene soddisfatta dall'inserimento della stringa *linein* al posto di *%s*.

Se il nome del file può essere copiato senza problemi, allora si procede con l'apertura del file dalla riga 14 alla 20. Se la chiamata `open` non ha successo, restituisce un valore negativo che provoca il salto all'etichetta *9* della riga 28 per la stampa di un messaggio d'errore. Altrimenti la `open` restituisce il descrittore di file che viene memorizzato nella variabile *fildes*. Il descrittore di file serve per le successive chiamate di `read` e `lseek`.

Poi si procede con la lettura del file a blocchi di 512 byte, memorizzati nel buffer *buf*. Il buffer è stato allocato in un'area di memoria che eccede di due byte la dimensione di blocco per puro scopo dimostrativo: la dimensione può essere specificata tramite un'espressione che combina una costante simbolica a un intero (riga 123). Analogamente, alla riga 21 viene assegnato a `SI` il secondo elemento dell'array *linh*, perciò resta una parola di valore 0 tra l'inizio dell'array e la locazione puntata da `SI`. Il registro `BX` serve a contenere la locazione del primo carattere del file non ancora letto, perciò viene inizializzato a 0 prima che il buffer venga riempito alla riga 22.

Il riempimento del buffer è gestito dalla routine *fillbuf* che occupa le righe dalla 83 alla 93. La chiamata di sistema, effettuata dopo l'inserzione degli argomenti sullo stack, salva in `AX` il numero di byte effettivamente letti. Questo valore viene copiato in `CX`, che serve a contenere il numero di caratteri non letti ancora presenti nel buffer. `BX` contiene invece la posizione nel file del primo carattere non letto, perciò alla riga 91 si somma `CX` a `BX`. La riga 92 assegna a `DI` la prima posizione del buffer per preparare la sua scansione in cerca del successivo carattere di fine riga.

In seguito al ritorno da *fillbuf*, la riga 24 verifica che sia stato letto davvero qualcosa. In caso contrario si esce dalla parte di lettura bufferizzata e si salta alla riga 25, dove comincia la seconda parte del programma.

Siamo pronti per la scansione del buffer. La riga 26 carica il simbolo \n in `AL`, la riga 27 ricerca questo valore nel buffer leggendolo iterativamente con il ciclo `REP SCASB`. Ci sono due condizioni di uscita dal ciclo: `CX` raggiunge il valore 0 oppure uno dei valori è un carattere di fine riga. Se il flag zero è asserito vuol dire che l'ultimo simbolo scandito è \n e che la posizione del simbolo corrente (quello successivo al carattere di fine riga) deve essere memorizzato nell'array *linh*. Quindi viene incrementato il contatore *count*, viene calcolata la posizione del file in base al contenuto di `BX` e `CX` stabilisce il numero di caratteri ancora disponibili (righe dalla 29 alla 31). Le righe dalla 32 alla 24 effettuano la memorizzazione vera e propria: il contenuto dei registri `SI` e `DI` viene scambiato prima e dopo l'istruzione `STOS` perché questa considera `DI` come destinazione e `SI` come sorgente. Le righe 35, 36 e 37 verificano se il buffer contiene altri dati da trattare e saltano in base al valore di `CX`.

Una volta raggiunto il carattere di terminazione del file, abbiamo costruito una lista completa contenente le posizioni del primo carattere di ogni riga. Dato che *linh* comincia con la parola 0, sappiamo che la prima riga comincia all'indirizzo 0, mentre la seconda riga comincia alla posizione contenuta nella parola che si trova in *linh* + 2, e così via. La lunghezza della riga *n* è data dall'indirizzo d'inizio della riga *n* + 1 meno la posizione iniziale della riga *n*.

La parte restante del programma serve per leggere il numero di una riga, copiare la riga nel buffer e visualizzarla tramite la chiamata `write`. L'array *linh* contiene tutte le informazioni necessarie, dato che il suo *n*-esimo elemento contiene la locazione iniziale della riga *n*. Se il numero di riga richiesto è 0 o se è maggiore del numero di righe presenti, il programma termina saltando all'etichetta *7*.

Questa porzione di programma comincia con la chiamata alla subroutine *getnum* alla riga 46. La routine legge una riga dallo standard input e la memorizza nel buffer *linein* (righe dalla 95 alla 103). Segue la preparazione della chiamata SSCANF, i cui argomenti vengono impilati in ordine inverso: si effettua il push di *curlin*, che può contenere un valore intero, quindi c'è il push dell'indirizzo della stringa di formattazione *numfmt* e infine viene impilato l'indirizzo del buffer *linein* contenente il numero in notazione decimale. La subroutine di sistema SSCANF inserisce in *curlin* il valore binario se l'operazione è possibile, altrimenti restituisce in AX il valore 0. La riga 48 effettua il test del valore restituito: in caso d'insuccesso il programma genera un messaggio d'errore saltando all'etichetta *8*.

Se la subroutine *getnum* restituisce in *curlin* un valore intero corretto, questo viene copiato in BX e poi valutato nelle righe dalla 49 alla 53 per verificare se corrisponde a un numero di riga valido; in caso contrario il programma termina con la chiamata di EXIT.

A questo punto bisogna individuare la terminazione della riga selezionata e il numero di byte da leggere: a tal fine moltiplichiamo BX per 2 per mezzo dello scorrimento verso sinistra SHL. La posizione nel file della riga selezionata viene copiata in AX alla riga 55, mentre la posizione della riga successiva viene conservata in CX perché verrà usata in seguito per calcolare il numero di byte della riga corrente.

L'accesso diretto a un file si realizza tramite la chiamata lseek cui va specificato l'offset della posizione cui si vuole accedere direttamente. L'offset è specificato in base all'inizio del file, perciò alla riga 57 viene impilato un argomento con valore 0. L'argomento successivo è l'offset nel file di tipo long (cioè un intero di 32 bit), dunque impiliamo prima una parola con valore 0 e poi il valore di AX (righe 58 e 59) per formare un intero di 32 bit. In seguito impiliamo il descrittore di file e il codice di LSEEK prima di effettuare la chiamata della riga 62. Il risultato di LSEEK è la posizione corrente del file, memorizzata nella coppia di registri DX:AX. Se questo numero ci sta tutto in una parola (il che è sicuramente vero per file più piccoli di 65.536 byte), l'indirizzo è contenuto in AX, perciò sottraendolo da CX otteniamo il numero di byte da leggere per copiare la riga nel buffer.

Il resto del programma è di facile comprensione. Le righe dalla 64 alla 68 leggono la riga del file, le righe dalla 70 alla 72 scrivono la riga sullo standard output (che ha descrittore di file 1). Si noti che, dopo il ripristino parziale della riga 69, lo stack contiene ancora il contatore *count* e l'indirizzo del buffer. La riga 73 ripristina definitivamente il puntatore allo stack e, saltando all'etichetta *3*, si è pronti per cominciare un'altra iterazione di *getnum*.

Ringraziamenti

L'assemblatore utilizzato in questa appendice fa parte della raccolta di strumenti "Amsterdam Compiler Kit", disponibile all'indirizzo web www.cs.vu.nl/ack. Ringraziamo coloro che hanno partecipato alla sua progettazione iniziale: Johan Stevenson, Hans Schaminee e Hans de Vries. Siamo debitori a Ceriel Jacobs che ha gestito lo sviluppo di questo pacchetto software e ci ha aiutati ad adattarlo più e più volte perché soddisfacesse i requisiti del nostro corso universitario. Siamo riconoscenti a Elth Ogston per la lettura del manoscritto e la verifica d'esempi ed esercizi.

Vogliamo ringraziare anche Robbert van Renesse, progettista del tracer di PDP-11, e Jan-Mark Wams, progettista del tracer di Motorola 68000. Molte delle loro idee sono entrate a far parte del progetto del tracer descritto in questa appendice. Inoltre, vogliamo ringraziare i tanti insegnanti e amministratori di sistema che ci hanno assistito durante la frequentazione dei molti corsi di programmazione in linguaggio assemblativo che abbiamo seguito negli anni passati.

Problemi

1. Dopo l'esecuzione dell'istruzione MOV AX, 702, qual è il valore (decimale) contenuto nei registri AH e AL?
2. Il registro CS contiene il valore 4. Qual è l'intervallo d'indirizzi di memoria assoluti occupato dal segmento di codice?
3. Qual è il più grande indirizzo di memoria accessibile dall'8088?
4. Sia CS = 40, DS = 8000 e IP = 20.
   a. Qual è l'indirizzo assoluto dell'istruzione successiva?
   b. Se viene eseguita MOV AX, (2), quale parola di memoria viene caricata in AX?
5. Si immagini una subroutine con tre argomenti che viene chiamata con l'usuale sequenza: il chiamante impila i tre argomenti sullo stack in ordine inverso e quindi esegue l'istruzione CALL. Il chiamato salva il vecchio BP e inizializza il nuovo BP in modo che punti al suo vecchio valore. Quindi decrementa il puntatore allo stack in modo da allocare spazio sufficiente per le variabili locali. Date queste convenzioni, si indichi l'istruzione necessaria per copiare il primo argomento della subroutine nel registro AX.
6. Nella Figura C.1 viene usata l'espressione *de – hw*, cioè la differenza di due etichette, come operando di un'istruzione. È possibile immaginare una circostanza in cui sarebbe sensato usare come operando la somma di due etichette, per esempio *de + hw*? Si motivi la risposta.
7. Si scriva il codice assemblativo per il calcolo dell'espressione:
   ```
   x = a + b + 2
   ```
8. Una funzione C viene invocata con l'istruzione:
   ```
   foobar(x, y);
   ```
   Si scriva il codice assemblativo che realizza la stessa chiamata.
9. Si scriva un programma in linguaggio assemblativo che accetti in ingresso espressioni contenenti un intero, un operando e un altro intero, e che stampi in uscita il valore dell'espressione. Il programma deve gestire gli operatori +, −, × e /.

Indice analitico

A

8051 (*si veda* Intel 8051)
8088 (*si veda* Intel 8088)
Access Control List (ACL), 475
Accumulatore, 17, 18, 315, 338
ACK (*si veda* Amsterdam Compiler Kit)
ACL (*si veda* Access Control List)
ADSL (*si veda* Asymmetric DSL)
Advanced Microcontroller Bus Architecture (AMBA), 547
Advanced Programmable Interrupt Controller (APIC), 188
AGP, bus, 106, 200
Aiken, Howard, 13
Albero, 594
Algebra booleana minimale, 134
Algebra di Boole, 134-136
Algoritmi, 7
Algoritmo di allattamento a richiesta, 423
Algoritmo First-In First-Out (FIFO), 425
Alpha, 24
  DEC, 24
ALU (*si veda* Arithmetic Logic Unit)
AMBA (*si veda* Advanced Microcontroller Bus Architecture)
American Standard Code for Information Interchange (ASCII), 123-125
Ampiezza del bus, 173-174
Analytical engine, 13
Anello, 595
API di Win32, 462
APIC (*si veda* Advanced Programmable Interrupt Controller)
Apparecchiature per le telecomunicazioni, 113-121
Apple II, 23
Apple Macintosh, 24
Application Programming Interface (API), 462
Application-Specific Integrated Circuit (ASIC), 551
Approccio strutturale, 2
Arbitraggio del bus, 179-181
Arbitraggio del bus PCI, 202
Arbitro del bus, 100
Architettura, 7, 53
Architettura a tre bus, 267-268
Architettura degli elaboratori, 7
  pietre miliari, 12-25
Architettura Harvard, 77
Architettura load/store, 335
Architetture di multiprocessori simmetrici UMA, 573-581
Architetture per il calcolo parallelo, 529-628
  coprocessori, 548-561
  multicomputer,591-625
  multiprocessori, 562-563
  multiprocessori in un solo chip, 543-548
  multithreading, 537-543
  parallelismo a livello delle istruzioni, 531-537
  parallelismo nel chip, 531-548
  processori di rete, 548-555
  processori multimediali, 556-561
  tassonomia, 566-567
Architetture superscalari, 60-62
Area dei metodi, 241
Arithmetic Logic Unit (ALU), 5, 50, 151
Aritmetica binaria, 655-664
  conversione tra basi, 658-661
  negativa, 662-663
Aritmetica satura, 534
Array logici programmabili, 146-148
Arrotondamento, 669
as88, un assemblatore basato su ACK, 700-703

ASCII (*si veda* American Standard Code for Information Interchange)
ASIC (*si veda* Application-Specific Integrated Circuit)
Assemblaggio di memoria, 77-78
Assemblatore, 7, 492, 505-513, 678, 699-704
  a due passate, 505-506
  prima passata, 506-510
  seconda passata, 510-511
  tabella dei simboli, 511-513
Assemblatori a due passate, 505-506
Asymmetric DSL (ADSL), 116
AT Attachment (ATA-3), 84
ATA Packet Interface (ATAPI-4), 84
ATA-3, 84
Atanasoff, John, 13, 17
ATAPI-4, 84
Attesa attiva, 373
Autodecremento, 689
Autoincremento, 689
Automa a stati finiti per la predizione dei salti, 295
  Instruction Fetch Unit (IFU), 269
Auto-modificante, programma, 353

B

Babbage, Charles, 14
Banda larga, 116
Bardeen, John, 18
Base, 132, 657
  conversione tra, 658-661
  sistemi di numerazione in base fissa, 657-658
Basic Input Output System (BIOS), 83
Baud, 115
BCD (*si veda* Binary Coded Decimal)
Bechtolsheim, Andy, 41
Benchmark, 494
Best fit, algoritmo, 430
Binaria, aritmetica, 664
Binary Coded Decimal (BCD), 66, 686
BIOS (*si veda* Basic Input Output System)
Bipolare, tecnologia, 133
Bit, 66, 658
Bit di parità, 71
Bit map, 340
Bit presente/assente, 420
Bit slice, 152
Bitblt, 560
Blocchi di memoria, 419
Blocco delle variabili locali, 238, 240
Blocco elementare, 301
Blocco indiretto, 472
Blocco indiretto doppio, 472
Blocco indiretto triplo, 472
BlueGene/L, 596-600, 603-604
Blu-Ray, 98
Boole, George, 134
Branch Target Buffer (BTB), 306
Brattain, Walter, 18
BSS (*si veda* Block Started by Symbol)
BTB delle tracce, 307
BTB L1, 306
Buffer circolare, 448
Buffer di prefetch, 59
Buffer invertente, 162
Buffer non invertente, 162
Bug del millennio, 35
Burroughs B5000, 20
Bus, 19, 49, 99-102, 170-183, 196-214
  AGP, 106, 200
  asincroni, 177-179
  EISA, 198
  handshaking, 178
  ISA, 197
  multiplexato, 174
  PC IBM, 197
  PCI, 198-206
  PCI Express, 206-210
  Pentium 4, 188-190
  protocollo, 171
  sincroni, 174-177
  trasferimento di un blocco, 182
  USB, 210-214
Bus delle periferiche, 547
Bus di sistema, 171
Bus, disallineamento del, 174
Bus, master, 172
Byron, Lord, 14
Byte, 68, 327, 686
Byte di prefisso, 259, 348, 378

C

Cache, write-deferred, 291
Cache a blocchi, 457
Cache a corrispondenza diretta, 287
Cache dei dati, 311
Cache delle tracce, 306
Cache di prefetch, 311
Cache di secondo livello, 286
Cache hit, 288
Cache miss, 288
Cache Only Memory Access (COMA), multiprocessori, 567, 590
Cache set-associativa a n vie, 289
Cache set-associativa, 289-291
Cache specializzata, 77, 286
Cache unificata, 76-77
Cache, write-allocation, 291, 575

Cache, write-back, 291, 576
Cache, write-through, 290, 574
Calcolo che utilizza il parallelismo esplicito, 402
Call gate, 436
Caricamenti speculativi, 407-408
Caricamento, 513-525
Catamount, 603
Cathode Ray Tube (CRT), 102-103
CCD (*si veda* Charge-Coupled Device)
CC-NUMA (*si veda* Cache Coherent NUMA multiprocessors)
CD registrabili, 94
CD riscrivibili, 96
CDC (*si veda* Control Data Corporation)
CDC 6600, 13, 20, 63, 297, 548
CDC Cyber, 68
CD-R, 94
CD-ROM (*si veda* Compact Disc-Read Only Memory)
Celeron, 38
Celle (locazioni) di memoria, 67
Central Processing Unit (CPU), 18, 49
Centri di calcolo, 22
Charge-Coupled Device (CCD), 121
Checkerboarding, 429
Chiamata di macro, 501
Chiamate al supervisore, 10
Chiamate di sistema, 10, 415
Chiamate di sistema e subroutine di sistema, 696-698
Chiamate di subroutine, 695-696
Chiave, 442
Chip, 141
Chip della CPU, 168-170
Chip di memoria, 163-165
Chip RFID, 29
Chiusura, 656
Cicli di bus, 174
Cicli e operazioni iterative su stringhe, 692-693
Ciclo del percorso dati, 51
Ciclo di clock, 152
Ciclo esecutivo delle istruzioni, 52
Ciclo locale, 116
Ciclo virtuoso, 27
Cilindro, 81
Circuiti integrati, 21, 141-143
Circuiti per l'aritmetica, 148-152
Circuiti virtuali, 209
Clock, 152-154
Clone, 24
Cluster
  Google, 605-609
  NTFS, 477
  workstation, 34, 604, 604-609
Cluster di computer, 34-35, 604-609
Cluster di Google, 605-609
CLUT (*si veda* Color Look Up Table)
COBOL, programma, 35
Code di messaggi, 478
Code page, 125
Code point, 125
Codice di escape, 348
Codice operativo (opcode), 224
Codice operativo espandibile, 345-347
Codice Reed-Solomon, 79
Codici correttori, 70-74
Codici di condizione, 329
Codifica dei caratteri, 123-127
Codifica hash, 513
Coerenza della cache, 574
Collegamento, 469
  a tempo del binding, 518-521
  dinamico, 521-525
Collegamento a festone (daisy chaining), 179
Collegamento esplicito, 524
Collegamento implicito, 524
Collettore, 132
Collettore aperto, 172
Color Look Up Table (CLUT), 559
Colore indicizzato, 106, 557
COLOSSUS, 13, 16
COMA (*si veda* Cache Only Memory Access)
COMA semplice, 590
Comandi del tracer, 706-708
Commodity Off The Shelf (COTS), 34
Commutatori crossbar, 578-579
Commutazione a livello, 157
Commutazione di pacchetto, 549
Commutazione di pacchetto store-and-forward, 549
Commutazione sul fronte, flip-flop, 157
Compact Disc-Read Only Memory (CD-ROM), 91
  riscrivibile, 99
  settore, 92
  traccia, 94
  XA, 94
Compagnie telefoniche, 116
Comparatori, 146
Compilatore, 7, 492
Complemento a due, 662
Complemento a uno, 662
Computer con un insieme ridotto d'istruzioni, 56
  principi di progettazione, 57-58
  rispetto a CISC, 56

Computer "invisibili", 25
Computer con un insieme d'istruzioni complesso, 56
Computer di prima generazione, 15-18
Computer di quarta generazione, 22-24
Computer di seconda generazione, 18-20
Computer di terza generazione, 20-22
Computer integrati (embedded), 25
Computer per giocare, 32
Computer, tipologie, 26-35
Comunicatore, 611
Condivisione a soglia, 541
Condivisione ripartita delle risorse, 541
Condivisione totale delle risorse, 541
Confronti e salti condizionati, 369-370
Confronto tra architetture, 318
Consistenza debole, 571-572
Consistenza di processore, 571
Consistenza dopo rilascio, 572-573
Consistenza sequenziale, 570
Consistenza stretta, 569
Consumatore, 448
Contatti, 168
Contesto, 437
Control Data Corporation (CDC), 20
Controllo del flusso, a livello ISA, 384-396
  coroutine, 390-392
  diramazioni, 384-385
  interrupt, 393-396
  procedure, 385-390
  trap, 392
Controllo di flusso, 209
Controllore, 99
Controllore del disco, 82
Conversione tra basi, 658-661
Copia dopo scrittura, 465
Coprocessori, 548-561
Coprogettati, hardware e software, 25
Core, 543
CoreConnect, 546
Coroutine, 390-392
Corsa critica, 448-451
Corsia, 208
Costante di rilocazione, 517
COTS (*si veda* Commodity Off The Shelf)
COW (*si veda* Cluster Of Workstations)
CP/M, 23
CPP (*si veda* Constant Pool Pointer)
CPU (*si veda* Central Processing Unit)
CPU TriMedia, 532-537
Cray, Seymour, 20
Cray-1, 20, 54
CRC (*si veda* Cyclic Redundancy Check)
Creazione dei processi, 447
Crittografia a chiave pubblica, 561
Crittografia a chiave simmetrica, 561
Crittoprocessori, 561
Crosspoint, 578
CRT (*si veda* Cathode Ray Tube)
Cubo, 595
Cut through virtuale, 599
Cyclic Redundancy Check (CRC), 209, 213, 550

D

Dati scaduti, 574
DDR (*si veda* Double Data Rate memory)
De Morgan, legge, 139
DEC (*si veda* Digital Equipment Corporation)
DEC PDP-1, 19
DEC PDP-8, 19
DEC VAX, 54, 56
Decodifica dell'indirizzo, 215-218
Decodifica parziale dell'indirizzo, 217
Decodificatori, 145
Definizione di macro, 501
Demultiplexer, 145
Descrittore di file, 467, 724
Descrittore di sicurezza, 475
Destinatario, PCI, 201
Diametro, rete, 593
Difference engine (macchina differenziale), 14
Differenze tra assemblatori dell'8088, 703-704
Digital Equipment Corporation (DEC), 18, 20, 54
Digital Subscriber Line, 115-118
Digital Subscriber Line Access Multiplexer (DSLAM), 118
Dimensionalità, 594
Dimensione di pagina, 425-426
DIMM (*si veda* Dual Inline Memory Module)
Dinamico, collegamento, 521-525
DIP (*si veda* Dual Inline Package)
Dipendenza di tipo WAR, 299
Dipendenza effettiva, 279
Dipendenza RAW, 279
Dipendenza WAW, 299
Diramazioni (salti), 384-385
Direct Memory Access (DMA), 100, 374
Directory, 445
Directory delle pagine, 434
Directory di lavoro, 469
Directory radice, 470, 471
Direttive dell'assemblatore, 498
Diretto, indirizzamento, 352, 687

Disallineamento del bus, 174
Dischetto o floppy disk, 83
Dischi, 79-98
  CD-ROM, 91
  DVD, 96-98
  IDE, 83-85
  magnetici, 79-81
  ottici, 96-99
  RAID, 86-90
  SCSI, 85-86
  Winchester, 80
Disco digitale versatile, 96
Disco video digitale, 96
Dispositivi a tre stati, 162
Distanza di Hamming, 70
Distributed Shared Memory (DSM), 566, 613-615
Divisore, 117
DLL (*si veda* Dynamic Link Library)
DMA (*si veda* Direct Memory Access)
Double, 682
Double Data Rate Memory (DDR), 167
Double PoinTeR (DPTR), 316
Dpi (punti per pollice), 109
DRAM (*si veda* Dynamic RAM)
Driver del bus, 172
Driver di dispositivo, 461
DSL (*si veda* Digital Subscriber Line)
DSLAM (*si veda* Digital Subscriber Line Access Multiplexer)
DSM (*si veda* Distributed Shared Memory)
DSM hardware, 581
Dual Inline Package (DIP), 142
Duali, 139
Dump di memoria, 9
Dynamic Link Library (DLL), 523

E

Eckert, J. Presper, 16
ECL (*si veda* Emitter-Coupled Logic)
EDO (*si veda* Extended Data Output)
EDVAC (*si veda* Electronic Discrete Variable Automatic Computer)
EEPROM (*si veda* Electrically Erasable PROM)
EHCI (*si veda* Enhanced Host Controller Interface)
EIDE (*si veda* Extended IDE)
EISA (*si veda* Extended ISA)
Elaborazione in uscita dei pacchetti di rete, 553
Electrically Erasable PROM (EEPROM), 167
Electronic Discrete Variable Automatic Computer (EDVAC), 16
Electronic Numerical Integrator And Computer (ENIAC), 16
Emettitore, 132
Emulazione, 21-22
Enable, 156
Endian, 68
  big, 68
  little, 68
Enhanced Host Controller Interface, 213
ENIAC (*si veda* Electronic Numerical Integrator And Computer)
ENIGMA, 15
EPIC (*si veda* Explicitly Parallel Instruction Computing)
Epilogo della procedura, 388
EPROM (*si veda* Erasable PROM)
Equivalenza di circuiti, 138-141
Erasable Programmable ROM (PROM), 43, 167
Errore di pagina, 422
Errore di underflow, 669
Errore relativo, 669
Esadecimali, 657, 701
Esecutiva, Windows XP, 461
Esecuzione condizionata, 405
Esecuzione dell'istruzione, 51-55
Esecuzione fuori sequenza, 296-301
Esecuzione speculativa, 301-304
Espansione di macro, 501
Esponente, 668
Estensione del segno, 230
Estridge, Philip, 23
Ethernet, 548
Etichette, 678
Eventi, 483
Evoluzione delle macchine multilivello, 7-12
Extended Data Output (EDO), 166
Extended IDE (EIDE), 84
Extended ISA (EISA), 101, 198

F

Falsa condivisione, 615
Famiglia MCS-51, 43
Fanout, 592
Fast Page Mode Memory (FPM), 166
FAT (*si veda* File Allocation Table)
Fat tree, 594
Fetch-esecuzione, 224
Fibre, 482
Field Programmable Gate Array (FPGA), 551
FIFO (*si veda* First-In First-Out algorithm)
File, 440-442
File Allocation Table (FAT), 473
File immediato, 477
File system
  UNIX, 467-473
  Windows XP, 473-477

Filtri, 468
Filtro di Bayer, 121
Finestra di registri, 334
Firewall, 549
First fit, algoritmo, 430
Flag della modalità d'incremento, 689
Flip-flop, 157-159
Floppy disk o dischetto, 83
Formati d'istruzione, 342-351
  8051, 350-351
  criteri progettuali, 343-345
  livello ISA, 343-351
  Pentium 4, 348-349
  UltraSPARC, 349-350
Forrester, Jay, 18
FORTRAN, 9-10
FORTRAN Monitor System (FMS), 9
FPGA (*si veda* Field Programmable Gate Array)
FPM (*si veda* Fast page Mode Memory)
Frame, 92
Frammentazione
  esterna, 429
  interna, 425
Frazione, 668
Frequenza del retino dei mezzitoni, 111
Frequenza di fallimento (miss ratio), 75
Frequenza di successi (hit ratio), 75
FSM (*si veda* Finite State Machine)
Full handshake, 178
Full-duplex, 115
Furto di cicli, 100, 375

G

Gamut (gamma dei colori), 112
GDT (*si veda* Global Descriptor Table)
Gerarchie di memoria, 78-79
Gestione dei processi
  UNIX, 477-480
  Windows XP, 481-483
Gestore degli oggetti (Object manager, Windows XP), 461
Gestore dei processi e dei thread, Windows XP, 461
Gestore dell'interrupt, 100
Gestore della cache dei file, Windows XP, 461
Gestore della memoria virtuale, Windows XP, 461
Gestore di I/O, Windows XP, 461
Gestore di trap, 392
Gettone di accesso, 475
Giustificata a destra, parola, 366
Global Descriptor Table (GDT), 432
Globe, 619-620
Goldstine, Herman, 17
Grado (uscente), 592
Graphical User Interface (GUI), 23, 458
Griglia (grid), 595, 626-628
Gruppi d'istruzioni, 404
GUI (*si veda* Graphical User Interface)

H

Half adder (semisommatore), 149
Half-duplex, 115
Hamming, codice, 71
Hamming, Richard, 28
Handle, 463
Hardware, 7
  in relazione al software, 7
Hashing, 513
Hazard (rischio), 279
HDTV (*si veda* High Definition TeleVision)
Headless workstation, 605
High Definition TeleVision (HDTV), 560
High Sierra, 93
Hoagland, Al, 27
Hoff, Ted, 36
HTTP (*si veda* HyperText Transfer Protocol)
Hub principale, USB, 211
Hypertext Transfer Protocol (HTTP), 550
Hyperthreading, 540

I

I/O (*si veda* Input/Output)
I/O mappato in memoria, 215
IA-32, 331
IA-64, 400-408
  caricamenti speculativi, 407-408
  EPIC, modello, 409
  pacchetti d'istruzioni (bundle), 404
  scheduling delle istruzioni, 404-405
IAS, macchina, 17
IBM, 18, 19-20, 197
IBM 1401, 19, 21
IBM 360, 21, 25, 548
IBM 701, 18
IBM 704, 18
IBM 7094, 19, 25
IBM 801, 55
IBM PS/2, 197
IC (*si veda* Integrated Circuit)
IDE (*si veda* Integrated Drive Electronics)

IFU (*si veda* Instruction Fetch Unit)
IJVM, 223, 238-246
  area dei metodi, 240
  blocco delle variabili locali, 240
  codice Java, 246
  implementazione con Mic-1, 251-263
  implementazione con Mic-2, 272-275
  implementazione con Mic-4, 281-284
  insieme d'istruzioni, 242-245
  modello della memoria, 240-241
  operazioni della memoria, 229-230
  percorso dati, 224
  porzione costante di memoria, 240
  realizzazione di Mic-3, 276-283
  stack, 238-240
  stack degli operandi, 240
  temporizzazione, 227-229
ILC (*si veda* Instruction Location Counter)
ILLIAC, 16
ILLIAC IV, 63, 567
Immediato, indirizzamento, 351, 688
In linea, 445
Inchiostro a base di coloranti, 112
Inchiostro a pigmenti, 112
Indice destinazione, 683
Indice sorgente, 683
Indirizzamento, 342, 351-364
  8051, 363-364
  8088, 686-689
  a registro, 352
  a registro con indice, 688
  a registro indiretto, 352-353, 687
  a stack, 355-358
  con spiazzamento, 688
  diretto, 352
  immediato, 351
  implicito, 688
  indicizzato, 353-355
  indicizzato esteso, 355
  indiretto, 355, 687
  Pentium 4, 361-363
  UltraSPARC, 363
Indirizzi di memoria, 67-68
Indirizzi effettivi, 688
Indirizzo, 67, 680
  di memoria, 67-68
Indirizzo lineare, 433
Industry Standard Architecture (ISA), 101, 197
Infissa, notazione, 355
Iniziatore, bus PCI, 201
I-node, 471
Input/Output, 99-127
Input/Output, dispositivi apparecchiature per le telecomunicazioni, 113-121
  dischi magnetici, 79-81
  dischi ottici, 96
  macchine fotografiche digitali, 121-123
  modem, 114-115
  monitor CRT, 105-106
  mouse, 106-108
  schermi piatti, 103-105
  stampanti, 108-113
  tastiere, 102
  terminali, 102-106
Input/Output programmato, 372
Insiemi d'istruzioni a confronto, 381-384
Instruction Location Counter (ILC), 506
Instruction pointer, 680, 683
Instruction Register (registro istruzioni, IR), 49-50, 315
Instruction Set Architecture (ISA), 6, 677
  8051, 336-339
  Pentium 4, 330-332
  UltraSPARC, 333-336
Integer Unit (IU), 42
Integrated Drive Electronics (IDE), 83
Integrazione a grandissima scala, 22
Intel, 36
Intel 4004, 36
Intel 8008, 36
Intel 8051, 194-196
  formati d'istruzione, 350-351
  indirizzamento, 363-364
  istruzioni, 381-384
  microarchitettura, 315-317
  panoramica del livello ISA, 336-339
  storia, 42-44
  tipi di dati, 342
Intel 8080, 36
Intel 8086, 36-37
Intel 8088, 23, 36-37, 330, 677
  indirizzamento, 686-689
  organizzazione di memoria, 684-689
Intel 8255A, 214
Intel 80286, 37
Intel 80386, 37
Intel 80486, 37
Intel Celeron, 38
Intel Pentium 4 (*si veda* Pentium 4)
Intel Xeon, 40
Interconnessione completa, 594
Interfaccia a scambio di messaggi, 610-612
Interfaccia del dispositivo grafico, Windows XP, 462
Interfaccia di sistema, 461
Interlacciati, segnali, 559
Interlacciato, sistema, 581
Interna, frammentazione, 425

Internet Service Provider (ISP), 549
Internet via cavo, 118-121
Interpretazione, 2-4
Interprete, 2, 52, 678
Interrupt, 100, 393-396
  impreciso, 299
  preciso, 299
  trasparenza, 394
Intestazione, 207
Intestazione IP, 550
Invalidazione, strategia, 575
Invertitori, 132
Invocazione di procedura, 370-371
I/O virtuale, 440-446
  implementazione, 442-445
  UNIX, 467-473
  Windows XP, 473-477
IP, protocollo, 550
Ipercubo, 595
IR (*si veda* Instruction Register)
Iron oxide valley, 27
ISA (*si veda* Industry Standard Architecture)
ISA (*si veda* Instruction Set Architecture)
ISP (*si veda* Internet Service Provider)
Istruzioni, 330, 365-384
  8051, 381
  8088, 689-699
  a confronto, 369-370
  binarie, 366-367
  di ciclo, 371-372
  di I/O, 372-375
  di salto, 369-370, 384-385
  di salto e di chiamata, 693-695
  di trasferimento dati, 365-366
  di un byte, 686
  di una parola, 686
  invocazione di procedura, 370-371
  Pentium 4, 375-378
  UltraSPARC, 378-381
  unarie, 367-368
Itanium 2, 402
IU (*si veda* Integer Unit)

J

Java, 7
Java Virtual Machine, 223
Jobs, Steve, 23
JOHNIAC, 16
Joint Photographic Experts Group (JPEG), 123
Joint Test Action Group (JTAG), 560
Joy, Bill, 41
JPEG (*si veda* Joint Photographic Experts Group)
JTAG (*si veda* Joint Test Action Group)
JVM (*si veda anche* IJVM)
JVM (*si veda* Java Virtual Machine)

K

Kernel, 461
Khosla, Vinod, 41
Kildall, Gary, 23

L

LAN (*si veda* Local-Area Network)
Land, 96
Larghezza di banda complessiva, 621
Larghezza di banda del processore, 60
Larghezza di banda di bisezione, 594
Laser, stampante, 110
Latch, 154-156
Latch D temporizzato, 156
Latch SR temporizzato, 155-156
Latch SR, 154
Latenza, 60
  nascosta, 625
  rotazionale, 81
Latin-1, 125
LBA (*si veda* Logical Block Addressing)
LCD (*si veda* Liquid Crystal Display)
LDT (*si veda* Local Descriptor Table)
Least Recently Used Algorithm (LRU), 289, 424
LED (*si veda* Light Emitting Diode)
Legacy bus, 185
Legati debolmente, 530
Legati strettamente, 530
Legge di Amdahl, 623
Leibniz, Gottfried, 14
Letterali, 508
Libreria condivisa, 524-525
Libreria destinazione, 525
Libreria importata, 524
Libro arancione, 94
Libro giallo, 91
Libro rosso, 90
Libro verde, 93
Light Emitting Diode (LED), 107
Linda, 615-617
Linea di cache, 76, 287, 574
Linguaggi ad alto livello, 7
  e linguaggio assemblativo, 492-493, 495
Linguaggio assemblativo, 492-501, 677-726
  caratteristiche, 492-493
  istruzioni, 495-498
  motivazioni all'uso, 493-495
  pseudoistruzioni, 498-501
Linguaggio destinazione, 491

Linguaggio macchina, 1, 677
Linguaggio sorgente, 491
Linkage editor, 513
Linker, 513-525, 726
Lista delle locazioni libere, 444
Livelli, 2-7
Livello applicativo, 627
Livello dei dispositivi, 4, 132
Livello del linguaggio assemblativo, 491-524
Livello di architettura dell'insieme d'istruzioni, 6
Livello di astrazione hardware, 460
Livello di microarchitettura, 5, 223-237
  esempi, 304-317
  memoria cache, 285-291
  predizione dei salti, 291-296
  progettazione, 272
Livello di raccolta, 627
Livello di transazione, 209
Livello di trasmissione, 209
Livello fisico, 208
Livello ISA, 6, 323-409
  controllo del flusso, 384-396
  formati d'istruzione, 342-351
  indirizzamento, 351-364
  panoramica, 325-338
  tipi d'istruzioni, 365-384
  tipi di dati, 339-342
Livello logico digitale, 4, 131-222
  bus, 170-183
  chip della CPU, 168-170
  circuiti, 141-155
  interfacce di I/O, 214-218
  memoria, 154-168
  porte logiche, 131-132
Livello macchina del sistema operativo, 5-6, 415-483
Livello risorse, 627
Livello software, 210
Livello struttura, 627
Local Descriptor Table (LDT), 432
Local-Area Network (LAN), 548
Località spaziale, 286
Località temporale, 286
Logica negativa, 141
Logica positiva, 141
Logical Block Addressing (LBA), 84
Long, 682, 686
Lovelace, Ada, 14
Lpi (linee per pollice), 111
LRU (*si veda* Least Recent Used)
Lunghezza del percorso, 264, 266-272
LV (*si veda* Local Variable pointer)

M

Macchina virtuale, 2
Macchine fotografiche digitali, 121-123
Macchine multilivello, 4, 7-12
Macintosh, Apple, 23
Macro con parametri, 503
Macro del sistema operativo, 10
Macroarchitettura, 238
Macroistruzioni, 501-505
  implementazione, 504-505
Mailslot, 482
Mainframe, 35
MAL (*si veda* Micro Assembly Language)
MANIAC, 16
Mantissa, 668
Mappa di memoria, 418
MAR (*si veda* Memory Address Register)
Mark I, 13, 15
Maschera, 366
Maschera del canale alpha, 559
MASM, 496
Massively Parallel Pprocessor (MPP), 595, 595-604
Master File Table (MFT), 476
Master, bus, 172
Matrice di aghi, stampante, 108
Mauchley, John 13, 15-16
MBR (*si veda* Memory Buffer Register)
McNealy, Scott, 41
MDR (*si veda* Memory Data Register)
Memoria, 66, 154-168
  8088, 684-689
  associativa, 432, 511-512
  condivisa a livello applicativo, 613-620
  d'attrazione, 590
  DDR, 167
  DRAM, 166
  EDO, 166
  EEPROM, 167
  EPROM, 167
  flash, 178
  FPM, 166
  non volatile, 167-168
  principale, 66-78
  PROM, 167
  RAM dinamiche, 166
  RAM statiche, 166
  SDRAM, 166
  secondaria, 78-98
  SRAM, 166
Memoria cache, 37, 74-77, 285-291, 574-576
  a corrispondenza diretta, 287-289
  di secondo livello, 286
  strategia d'invalidazione, 575
Memoria virtuale, 416-439
  e caching, 439
  Pentium 4, 432-436
  UltraSPARC III, 436-439
  UNIX, 463-465
  Windows XP, 465-467

Memorie di controllo, 55, 233
Memory Address Register (MAR), 229, 229
Memory Buffer Register (MBR), 229
Memory Data Register (MDR), 229, 229
Memory Management Unit (MMU), 419
Mescolamento perfetto, 580
Mesh, rete, 595
MESI, protocollo, 576-578
  separato, 286
  set-associativa, 289-291
  snooping, 574-576
  strategia di aggiornamento, 575
  write-allocation, 291, 575
  write-back, 291, 576
  write-deferred, 291
  write-through, 290
Messa a punto, 494
Metal Oxide Semiconductor (MOS), 133
Metodo, 370
Mezzitoni, 111
MFT (*si veda* Master File Table)
Mic-1, 233, 251-263
  implementazione, 251-263
  percorso dati, 234
Mic-2, 272-275
  percorso dati, 273
  realizzazione, 272-275
Mic-3, 276-283
  percorso dati, 277
  realizzazione, 276-283
Mic-4, 281-284
  percorso dati, 282
  realizzazione, 281-284
Mickey, 108
Microarchitettura
  8051, 315-317
  Pentium 4, 304-309
  UltraSPARC, 310-315
Microarchitettura NetBurst, 304-306
Micro Assemby Language (MAL), 248
Microcontrollori, 30-32
Microdrive, 123
MicroInstruction Register (MIR), 233
Microistruzioni, 55, 230-232
  notazione, 247-251
Micro-operazioni, 281
MicroProgram counter (MPC), 233
Microprogramma, 5, 680
Microprogrammazione, 5
  storia, 8-11, 22
Microsoft, 24, 458
MIMD (*si veda* Multiple Instruction-stream Multiple Data-stream)
MIPS (acronimo), 60
MIPS (chip), 56
MIR (*si veda* MicroInstruction Register)
MISD (*si veda* Multiple instruction stream single data stream computer)
MMU (*si veda* Memory Management Unit)
MMX (*si veda* MultiMedia eXtensions)
Modalità a registro, 352
Modalità d'indirizzamento, 351-364
  8051, 363-364
  a registro, 352
  analisi, 364
  diretto, 352
  immediato, 351
  indicizzato, 353-355
  indicizzato esteso, 355
  istruzioni di salto, 359
  Pentium 4, 361-363
  stack, 355-358
  UltraSPARC III, 363
Modalità kernel, 326
Modalità reale, 331
Modalità utente, 326
Modalità virtuale 8086, 331
Modem, 114-115
Modulazione, 114
  d'ampiezza, 114
  di fase, 115
  di frequenza, 115
Modulazione a coppia di bit, 115
Modulo e segno, 662
Modulo oggetto, 517-518
Monitor di sicurezza, Windows XP, 461
Moore, Gordon, 26
  legge di, 26-27, 40
MOS (*si veda* Metal Oxide Semiconductor)
MPP, 595-604
  BlueGene/L, 596-600
  Google, cluster, 605-609
  Red Storm, 600-603
Motif, 458
Motion Picture Experts Group (MPEG), 545
Motori di stampa, 111
Motorola 68000, 55
Mouse, 106-108
MPC (*si veda* MicroProgram Counter)
MPEG-2, 545
MPI (*si veda* Message-Passing Interface)
MPP (*si veda* Massively Parallel Processor)
MS-DOS, 24
Multicomputer a scambio di messaggi, 591-625
Multicomputer, prestazioni, 620-625
Multicomputer, scheduling, 612-613
MULTICS, 521
  moduli oggetto, 517-518
  UNIX, 524-525
  Windows, 523-524

MULTICS (*si veda* MULTiplexed Information and Computing Service)
MultiMedia eXtensions (MMX), 38
Multiplexato, bus, 174
MULTiplexed Information and computing service (MULTIX), 432, 521
Multiplexer, 143
Multiprocessore basato su directory, 582-583
Multiprocessori, 64-65, 312, 562-563, 591-625
  commutatori crossbar, 578-579
  orientati al bus, 574
  reti a commutazione, 579-581
Multiprocessori con memoria condivisa, 562-591
Multiprocessori in un solo chip, 543-548
Multiprocessori NUMA, 581-589
Multiprocessori NC-NUMA, 581-582
Multiprogrammazione, 21
Multisessione, 95
Multithreading a grana fine, 537
Multithreading a grana grossa, 538
Multithreading nel chip, 537-543
Multithreading simultaneo, 539
Multithreading, 537-543
Mutex, 479
Mutua esclusione, 480
Myhrvold, Nathan, 27

N

NaN (*si veda* Not a Number)
Nathan, prima legge del software, 27
Network Interface Device (NID), 117
Nexperia, 556-561
Nibble, 376
NID (*si veda* Network Interface Device)
No Remote Memory Access (NORMA), 569
Nomi mnemonici, 678, 699
Nomi simbolici, 678
Non in linea, 445
Nonuniform Memory Access (NUMA), 567
NORMA (*si veda* NO Remote Memory Access computer)
Not a Number, 674
Notazione in eccesso, 662
Notazione polacca, 355-357
Notazione polacca inversa, 355-357
NOW (*si veda* Network Of Workstations)
NT file system (NTFS), 473
NTFS (*si veda* NT File System)
NUMA (*si veda* NonUniform Memory Access)
Numeri a precisione finita, 655-657
Numeri denormalizzati, 673
Numeri in virgola mobile, 667-674

O

OCP-IP (*si veda* Open Core Protocol-International Partnership)
OGSA (*si veda* Open Grid Services Architecture)
OHCI (*si veda* Open Host Controller Interface)
Omnibus, PDP-8, 19
OPC (*si veda* Opcode register)
Open Core Protocol-International Partnership (OCP-IP), 547
Open Grid Services Architecture (OGSA), 628
Open Host Controller Interface, 213
Operando destinazione, 686
Operando immediato, 351
Operazioni (TRImedia), 532
Operazioni binarie, 366-367
Operazioni del bus, 181-183
Operazioni logiche, su bit e di scorrimento, 692
Opposto, 368
Orca, 617-619
OR-cablata, 172
Ordinamento dei byte, 68-70
Organizzazione della memoria, 160-163
Organizzazione virtuale, 626
OS/2, 24
Osborne-1, 24
Osservazioni sull'insieme d'istruzioni, 698-699
Ottali, 657, 728
Overlay, 416

P

Pacchetti d'istruzioni, 404
Pacchetto, 207, 549, 592
Pacchetto di ackowledgment (conferma), 209
Packet Processing Engine (PPE), 552
Pagina gemella, 615
Pagina impegnata (committed), 465
Pagina libera, 465
Pagina riservata, 465

Paginazione, 417-419
a richiesta, 422-423
implementazione, 419-422
Pagine, 418
Parallel Input/Output (PIO), 214
Parallel Virtual Machine (PVM), 610
Parallelismo
a livello d'istruzione, 58-62, 531-537
a livello di processore, 62-65
Parallelismo nel chip, 531-548
Parametri attuali, 503
Parametri formali, 503
Parola di codice, 70
Parole, 68, 686
Pascal, Blaise, 13
Passata, linguaggio assemblativo, 506
Passo elementare, 278
Payload, 212
PC IBM, 23, 27
bus, 197
origini, 23
PC (*si veda* Personal Computer)
PC (*si veda* Program Counter)
PCI (*si veda* Peripheral Component Interconnect)
PDA (*si veda* Personal Digital Assistant)
PDP-1, 19
PDP-11, 22
PDP-8, 19
Pentium 4, 184-190
bus, 188-190
contatti, 186-188
formati d'istruzione, 348-349
indirizzamento, 361-363
introduzione, 36-41
istruzioni, 375-378
memoria virtuale, 432-436
microarchitettura, 304-309
panoramica del livello ISA, 330-332
problemi, 401-402
storia, 38-41
tipi di dati, 341
torri di Hanoi, 397
Percorsi relativi, 469
Percorso, 469
Percorso assoluto, 469
Percorso dati, 5, 50, 224
Mic-1, 233
Mic-2, 273
Mic-3, 276
Mic-4, 282
temporizzazione, 227-229
Percorso dati a pipeline, 277
Peripheral Component Interconnect (PCI), 101, 198-206
segnali, 203-205
transazioni, 205-206
Personal computer, 34
Personal Digital Assistant (PDA), 25
Pervasive computing (computazione pervasiva), 25
Pietre miliari nell'architettura dei computer, 12-25
PIO (*si veda* Parallel Input/Output)
Pipe, 478
Pipeline, 63
a sette stadi, 281-284
Mic-3, 280
Mic-4, 292
Pentium 4, 304-309
Pentium I, 60-61
UltraSPARC, 312-315
Pipeline in stallo, 292
Pipeline U, 60
Pipeline V, 61
Pipeline, stadi, 59
Pipelining, 59
Pit, 96
Pixel, 106
PLA (*si veda* Programmable Logic Array)
Plain Old Telephone Service (POTS), 116
PlayStation 2, 32
Poison bit, 304
Politica di sostituzione delle pagine, 423-425
FIFO, 425
LRU, 424
Portable Operating System -IX (POSIX), 456
Portante, 114
Porte, 4, 131-141
Porzione costante di memoria, 240
POSIX (*si veda* Portable Operating System-IX)
Posizione di ritardo, 292
Postfissa, notazione, 355
POTS (*si veda* Plain Old Telephone Service)
PPE (*si veda* Protocol Processing Engine)
Preambolo, 79
Precisione doppia, 339
Predicati, 405-407
Predizione dei salti, 291-296
dinamica, 293-296
statica, 296
Prefetching, 625
Prestazioni, calcolatori paralleli, 620-625
legge di Amdahl, 623
miglioramento, 624-625
nascondere la latenza, 625
parametri di valutazione hardware, 620-621
parametri di valutazione software, 622-623
Principi di progettazione RISC, 57-58
Principio di località, 75, 423
Problema dei riferimenti in avanti, 506
Problema della rilocazione, 515
Procedure, 385-390

Procedure ricorsive, 385
Processi, 416
Processo di assemblaggio, 505-513
Processo figlio, 477
Processo genitore, 477
Processore vettoriale, 64
Processore, 546
Processori, 49-50
Processori di rete, 548-555
Processori multimediali, 556-561
Produttore e consumatore, problema, 448
Progetto giapponese di quinta generazione, 25
Program Counter (PC), 49, 224, 229, 316
Program counter, 680
Program Status Word (PSW), 315, 329, 435
Programma, 1
Programma binario, 678
Programma eseguibile binario, 491, 513
Programma indipendente dalla posizione, 521
Programma oggetto, 491
Programmable Processing Engine (PPE), 552
Programmable ROM (PROM), 167
Programmatori di sistema, 6
Programmazione in linguaggio assemblativo, 677-726
Prologo della procedura, 388
PROM (*si veda* Programmable ROM)
Protocol Processing Engine (PPE), 552
Protocolli Internet, 550
Protocollo di coerenza delle cache, 574
MESI, 576-578
Protocollo del bus, 171
Pseudoistruzioni, 498-501, 701, 709
PSW (*si veda* Program Status Word)
Pthread, 479
Puntatore, 352
Puntatore al record d'attivazione, 332
Puntatore alla porzione costante di memoria, 240
Puntatore allo stack, 683
Puntatore base, 683
Punti d'ingresso, 517
PVM (*si veda* Parallel Virtual Machine)

R

Radio Frequency Identification (RFID), 29
RAID (*si veda* Redundant Array of Inexpensive Disks)
RAM (*si veda* Random Access Memory)
RAM della scheda video, 105-106
RAM dinamiche, 166
RAM statiche, 166
Random Access Memory (RAM), 43, 166-168
RAM dinamiche, 166
RAM statiche, 166
synchronous DRAM (SDRAM), 166
Ranging, 120
Read-Only Memory (ROM), 40, 166-168
Record d'attivazione, 683
Record logici, 442
Red Storm, 600-603
Redundant Array of Inexpensive Disks (RAID), 86-90
Refresh (ricarica), 166
Registrazione perpendicolare, 80
Registri, 5, 159-160, 329-330, 680
Registri a scorrimento, 148-149
Registri dei codici di condizione, 684
Registri di periferica, 547
Registri di segmento, 684
Registri puntatore e registri indice, 683
Registro a scorrimento della storia dei salti, 295
Registro accumulatore, 681
Registro base, 681
Registro contatore, 682
Registro d'uso generale, 681
Registro dati, 682
Registro di flag, 329, 684
Registro di una parola, 686
Registro vettoriale, 64
Registro virtuale, 237
Recorder Buffer (ROB), 307
Replicated worker model, 617
Rete a commutazione multilivello, 579
Rete bloccante, 580
Rete omega, 579
Reti a commutazione, 579-581
Reti (circuiti)
combinatorie, 143-148
per l'aritmetica, 148-152
Reti combinatorie, 143-148
comparatori, 146
decodificatori, 145-149, 152
multiplexer, 143
Reti d'interconnessione, 592-595
larghezza di banda di bisezione, 594
topologia, 592-595
Reti di workstation, 581
Reti non bloccanti, 579
Retrocompatibile, 324
Ricerca dicotomica, 512
Ricevitore del bus, 172

Ricorsione, 370
Riferimenti esterni, 516
Rifornimento delle istruzioni, 310
Rilocazione dinamica, 518-521
Rinomina dei registri, 296-301
RISC (*si veda* Reduced Instruction Set Computer)
RISC contro CISC, 55-57
RISC principi di progettazione, 57-58
Ritardo della porta, 143
ROB (*si veda* ReOrder Buffer)
ROM (*si veda* Read-Only Memory)
Router, 549
Routine di servizio dell'interrupt, 394

S

Scalable Processor ARChitecture (SPARC), 41
Scambio di messaggi, 610
Scambio di messaggi bufferizzato, 610
Scambio di messaggi non bloccante, 610
Scambio sincrono di messaggi, 610
Scanner delle pagine, 582
Scansione progressiva, 559
Scansione raster, 103
Scheda madre, 99
Schedulatori, 308
Scheduling delle istruzioni, 404-405
Scheduling, multicomputer, 612-613
Schermi a matrice attiva, 105
Schermi a matrice passiva, 105
Schermi piatti, 103-105
Schermi TFT, 105
Schermo a cristalli liquidi (LCD), 103
Scoreboard, 297
SCSI (*si veda* Small Computer System Interface)
SDRAM (*si veda* Synchronous DRAM)
Seastar, 601
Secchielli (bucket), 513
Security ID (SID), 475
Seek (ricerca), 81
Segmentazione, 426-429
  algoritmo best fit, 430
  algoritmo first fit, 430
Segmento, 429, 684
Segmento di codice, 680
Segmento di collegamento, 521
Segnale asserito, 163
Segnale negato, 163
Segnali di controllo, 229
Selezione del riporto, 151
Semafori, 452-455
Semantica della memoria, 569-573
  consistenza debole, 571-572
  consistenza di processore, 571
  consistenza dopo rilascio, 572-573
  consistenza sequenziale, 570
Sequenzializzatore, 233
Serial ATA, 84
Server, 34
Server farm, 35
Servizi di sistema, Windows XP, 462
Sessioni, CD-ROM, 95
Settore, dischi, 79
Sezioni critiche, 483
Sezioni informative, 326
Sezioni normative, 326
Shard, 606
Shell, 458
Shockley, William, 18
SIB (*si veda* Scale, Index, Base)
SID (*si veda* Security ID)
Significando, 672
Simboli esterni, 518
SIMD (*si veda* Single Instruction-stream Multiple Data-stream)
SIMM (*si veda* Single Inline Memory Module)
Simplex, 115
Sincronizzazione dei processi, 452-455
Single Inline Memory Module (SIMM), 77
Single Instruction Stream Multiple Data Stream Computer, 63, 567
Single Large Expensive Disk (SLED), 86
Sistema a memoria distribuita, 563
Sistema batch, 10
Sistema big endian, 68
Sistema little endian, 68, 686
Sistemi a condivisione di tempo, 11
Sistemi operativi, 8, 415, 455-483
  CP/M, 23
  OS/2, 24
  storia, 8-12
  UNIX, 455-463, 463-465, 467-473, 477-480
  Windows, 458-463, 465-467, 473-477, 481-483
Slave, bus, 172
SLED (*si veda* Single Large Expensive Disk)
Slittamento, codice, 302
Small Computer System Interface (SCSI), 85-86
Small Outline DIMM (SO-DIMM), 77
SMP (*si veda* Symmetric MultiProcessor)
Snooping, cache, 185, 574-576
Socket, 457
SO-DIMM (*si veda* Small outline DIMM)

Software, 7
  di comunicazione, 609-612
  in confronto con l'hardware, 7
  scheduling, 612-613
Software di comunicazione per multicomputer, 609-612
Solaris, 456
Sommatore, 149
Sommatore a propagazione di riporto, 150
Sommatori (adder), 149-151
  a propagazione di riporto, 150
  half, 149
  selezione del riporto, 150
Sottosistema ambientale, Windows XP, 462
Sottosistema Win32, 462
SPARC (*si veda* Scalable PRocessor ARChitecture)
Spazio degli indirizzi, 417
Spazio degli indirizzi fisici, 418
Spazio degli indirizzi virtuali, 418
Spazio di tuple, 616
Spazio tra settori, 79
Spina a vampiro, 548
SRAM (*si veda* Static RAM)
SSE (*si veda* Streaming SIMD Extensions)
Stack, 238-240
Stack degli operandi, 223
Stadi, pipeline, 59
Stallo, 279
Stallo, pipeline in, 292
Stampante a getto di cera, 113
Stampante a sublimazione, 113
Stampanti, 108-113
  a colori, 112-113
  a inchiostro solido, 113
  a matrice di aghi, 108
  a sublimazione, 113
  laser, 110
  monocromatiche, 108-111
Stampanti a getto d'inchiostro, 109
Stampanti CYMK, 112
Standard error, 468
Standard in virgola mobile IEEE 754, 671-674
Standard input, 468
Standard output, 468
Stati
  8051, 318
  automa a stati finiti, 270
Stati di attesa, 176
Stazioni di testa, 118
Stella, 594
Stibbitz, George, 15
Storia
  1955-1965, 18-20
  1965-1980, 20-22
  1945-1955, 15-18
  1980-(?), 22-24
  dei computer, 12-25
  Intel, 36-41
  Sun Mirosystems, 41-42
Store-to-load, 309
Strategia di aggiornamento, 575
Strati, 3
Stream, 457
Streaming SIMD Extensions (SSE), 38
Striping, 87
Strobe, 156
Subroutine, 370, 721
Sun Fire E25K, 585-589
Sun Microsystems, 41-42
Sun UltraSPARC (*si veda* UltraSPARC)
Supercomputer, 20, 35
Superuser, 471
Symmetric MultiProcessor (SMP), 563

T

Tabella degli indici, 442
Tabella dei salti, 310
Tabella dei simboli, 506, 511-513
Tabella delle pagine, 418
Tabella di traduzione, 438
Tabella di verità, 134
Task bag, 617
Tassonomia dei calcolatori paralleli, 566-567
Tassonomia di Flynn, 566-567
Tastiere, 102
TAT-12/13, 27
Tavolozza, 106
TCP (*si veda* Transmission Control Protocol)
Telco, 116
Template, 616
Tempo del binding, 518-521
Temporizzazione del bus, 174-179
Thin film transistor, 105
Thrashing, 425
Thread, 479
Tipi di dati, 339-342
  non numerici, 340
  numerici, 339-340
Tipi d'istruzioni, 365-384
TLB (*si veda* Translation Lookaside Buffer)
TN (*si veda* Twisted Nematic)
Token, 546
Topologia virtuale, 612
Toro, 595
Torri di Hanoi, 396-400
  Pentium, 397
  UltraSPARC, 397-400
TOS (*si veda* Top Of Stack pointer)
Tracer, 678, 705
Traduttori, 491
Traduzione, 2-4
Transistor, invenzione dei, 18

Transistor-Transistor Logic (TTL), 133
Transizioni, 270
Translation Lookaside Buffer (TLB), 312, 437
  fallimento, 437
Translation Storage Buffer (TSB), 437
Transmission Control Protocol (TCP), 550
  intestazione, 550
Trap, 392, 716
Trasferimento di un blocco, bus, 182
Trasferimento, copia e aritmetica, 689-692
Trasmettitore-ricevitore del bus, 172
Trasparente (paginazione), 418
Trasparenza, 394
TSB (*si veda* Translation Storage Buffer)
TTL (*si veda* Transistor-Transistor Logic)
Tuple, 616
Twisted Nematic (TN), 104
TX-0, 18
TX-2, 18

U

UART (*si veda* Universal Asynchronous Receiver Transmitter)
Ubiquitous computing (computazione onnipresente), 25
UDB (*si veda* UltraSPARC, data buffer)
UHCI (*si veda* Universal Host Controller Interface)
Ultra Port Architecture (UPA), 192
UltraSPARC I, 42
UltraSPARC III, 190-195
  formati d'istruzione, 349-350
  indirizzamento, 363-
  istruzioni, 378-381
  memoria virtuale, 436-439
  microarchitettura, 310-315
  panoramica del livello ISA, 333-336
  storia, 41-42
  tipi di dati, 341
  torri di Hanoi, 397-400, 409
  UDB II, 192
UltraSPARC Data Buffer II, 192
UMA (*si veda* Uniform Memory Access)
UNICODE, 125-127
Uniform Memory Access (UMA), 567
Unità di accodamento, 281
Unità di allocazione/rinomina, Pentium 4, 307
Unità di controllo microprogrammata, 233-237
Unità di decodifica, 281
Unità di prelievo dell'istruzione, 268-272
Unità di ritiro, 309
Unità funzionali, 61
Unità metriche, 45
Universal Asynchronous Receiver Transmitter (UART), 214
Universal Serial Bus (USB), 210-214
Universal Synchronous Asynchronous Receiver Transmitter (USART), 214
UNIX
  collegamento, 524-525
  gestione dei processi, 477-480
  introduzione, 455-463
  memoria virtuale, 463-465
  servizidi I/O, 467-477
UPA (*si veda* Ultra Port Architecture)
USART (*si veda* Universal Synchronous Asynchronous Receiver Transmitter)
USB (*si veda* Universal Serial Bus)
USB 2.0, 213

V

Valore di sinistra dell'assegnamento, 686
Variabili di condizione, 480
VAX, 54
VCI (*si veda* Virtual Component Interconnect)
Very Long Instruction Word (VLIW), 531
Vettore di interrupt, 183, 393
Vettoriale, registro, 64
Virtual Component Interconnect (VCI), 547
VIS (*si veda* Visual Instruction Set)
Visual Instruction Set (VIS), 42
VLIW (*si veda* Very Long Instruction Word)
VLSI (*si veda* Very Large Scale Integration)
Volume Table Of Contents (VTOC), 94
Von Neumann, John, 17
  macchina di, 17
VTOC (*si veda* Volume Table Of Contents)

W

WAN (*si veda* Wide-Area Network)
Wattel, Evert, 677
WEIZAC, 16
Whirlwind I, 18

Wide-Area Network (WAN), 548
Wilkes, Maurice, 8, 54
Windows 95, 458
Windows 98, 459
Windows XP
  file, 473-477
  gestione dei processi, 481-483
  introduzione, 458-463
  memoria virtuale, 465-467
Windows New Technology (NT), 459
Windows NT (*si veda* Windows New Technology)
Windows, (*si veda anche* Windows XP) 24, 458
  collegamento in, 523-524
  storia, 24
Working set, 422-423
Wozniak, Steve, 23

X

X Windows, 458
Xeon, 40

Z

Zilog Z8000, 55
Zuse, Konrad, 15, 16

Prentice Hall License Agreement and Limited Warranty

READ THE FOLLOWING TERMS AND CONDITIONS CAREFULLY BEFORE OPENING THIS SOFTWARE PACKAGE. THIS LEGAL DOCUMENT IS AN AGREEMENT BETWEEN YOU AND PRENTICE HALL, INC. (THE "COMPANY"). BY OPENING THIS SEALED SOFTWARE PACKAGE, YOU ARE AGREEING TO BE BOUND BY THESE TERMS AND CONDITIONS. IF YOU DO NOT AGREE WITH THESE TERMS AND CONDITIONS, DO NOT OPEN THE SOFTWARE PACKAGE. PROMPTLY RETURN THE UNOPENED SOFTWARE PACKAGE AND ALL ACCOMPANYING ITEMS TO THE PLACE YOU OBTAINED THEM FOR A FULL REFUND OF ANY SUMS YOU HAVE PAID.

1. GRANT OF LICENSE: In consideration of your purchase of this book, and your agreement to abide by the terms and conditions of this Agreement, the Company grants to you a nonexclusive right to use and display the copy of the enclosed software program (hereinafter the "SOFTWARE") on a single computer (i.e., with a single CPU) at a single location so long as you comply with the terms of this Agreement. The Company reserves all rights not expressly granted to you under this Agreement.

2. OWNERSHIP OF SOFTWARE: You own only the magnetic or physical media (the enclosed media) on which the SOFTWARE is recorded or fixed, but the Company and the software developers retain all the rights, title, and ownership to the SOFTWARE recorded on the original media copy(ies) and all subsequent copies of the SOFTWARE, regardless of the form or media on which the original or other copies may exist. This license is not a sale of the original SOFTWARE or any copy to you.

3. COPY RESTRICTIONS: This SOFTWARE and the accompanying printed materials and user manual (the "Documentation") are the subject of copyright. The individual programs on the media are copyrighted by the authors of each program. Some of the programs on the media include separate licensing agreements. If you intend to use one of these programs, you must read and follow its accompanying license agreement. You may not copy the Documentation or the SOFTWARE, except that you may make a single copy of the SOFTWARE for backup or archival purposes only. You may be held legally responsible for any copying or copyright infringement which is caused or encouraged by your failure to abide by the terms of this restriction.

4. USE RESTRICTIONS: You may not network the SOFTWARE or otherwise use it on more than one computer or computer terminal at the same time. You may physically transfer the SOFTWARE from one computer to another provided that the SOFTWARE is used on only one computer at a time. You may not distribute copies of the SOFTWARE or Documentation to others. You may not reverse engineer, disassemble, decompile, modify, adapt, translate, or create derivative works based on the SOFTWARE or the Documentation without the prior written consent of the Company.

5. TRANSFER RESTRICTIONS: The enclosed SOFTWARE is licensed only to you and may not be transferred to any one else without the prior written consent of the Company. Any unauthorized transfer of the SOFTWARE shall result in the immediate termination of this Agreement.

6. TERMINATION: This license is effective until terminated. This license will terminate automatically without notice from the Company and become null and void if you fail to comply with any provisions or limitations of this license. Upon termination, you shall destroy the Documentation and all copies of the SOFTWARE. All provisions of this Agreement as to warranties, limitation of liability, remedies or damages, and our ownership rights shall survive termination.

7. MISCELLANEOUS: This Agreement shall be construed in accordance with the laws of the United States of America and the State of New York and shall benefit the Company, its affiliates, and assignees.

8. LIMITED WARRANTY AND DISCLAIMER OF WARRANTY: The Company warrants that the SOFTWARE, when properly used in accordance with the Documentation, will operate in substantial conformity with the description of the SOFTWARE set forth in the Documentation. The Company does not warrant that the SOFTWARE will meet your requirements or that the operation of the SOFTWARE will be uninterrupted or error-free. The Company warrants that the media on which the SOFTWARE is delivered shall be free from defects in materials and workmanship under normal use for a period of thirty (30) days from the date of your purchase. Your only remedy and the Company's only obligation under these limited warranties is, at the Company's option, return of the warranted item for a refund of any amounts paid by you or replacement of the item. Any replacement of SOFTWARE or media under the warranties shall not extend the original warranty period. The limited warranty set forth above shall not apply to any SOFTWARE which the Company determines in good faith has been subject to misuse, neglect, improper installation, repair, alteration, or damage by you. EXCEPT FOR THE EXPRESSED WARRANTIES SET FORTH ABOVE, THE COMPANY DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. EXCEPT FOR THE EXPRESS WARRANTY SET FORTH ABOVE, THE COMPANY DOES NOT WARRANT, GUARANTEE, OR MAKE ANY REPRESENTATION REGARDING THE USE OR THE RESULTS OF THE USE OF THE SOFTWARE IN TERMS OF ITS CORRECTNESS, ACCURACY, RELIABILITY, CURRENTNESS, OR OTHERWISE.

IN NO EVENT, SHALL THE COMPANY OR ITS EMPLOYEES, AGENTS, SUPPLIERS, OR CONTRACTORS BE LIABLE FOR ANY INCIDENTAL, INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF OR IN CONNECTION WITH THE LICENSE GRANTED UNDER THIS AGREEMENT, OR FOR LOSS OF USE, LOSS OF DATA, LOSS OF INCOME OR PROFIT, OR OTHER LOSSES, SUSTAINED AS A RESULT OF INJURY TO ANY PERSON, OR LOSS OF OR DAMAGE TO PROPERTY, OR CLAIMS OF THIRD PARTIES, EVEN IF THE COMPANY OR AN AUTHORIZED REPRESENTATIVE OF THE COMPANY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. IN NO EVENT SHALL LIABILITY OF THE COMPANY FOR DAMAGES WITH RESPECT TO THE SOFTWARE EXCEED THE AMOUNTS ACTUALLY PAID BY YOU, IF ANY, FOR THE SOFTWARE.

SOME JURISDICTIONS DO NOT ALLOW THE LIMITATION OF IMPLIED WARRANTIES OR LIABILITY FOR INCIDENTAL, INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES, SO THE ABOVE LIMITATIONS MAY NOT ALWAYS APPLY. THE WARRANTIES IN THIS AGREEMENT GIVE YOU SPECIFIC LEGAL RIGHTS AND YOU MAY ALSO HAVE OTHER RIGHTS WHICH VARY IN ACCORDANCE WITH LOCAL LAW.

ACKNOWLEDGMENT

YOU ACKNOWLEDGE THAT YOU HAVE READ THIS AGREEMENT, UNDERSTAND IT, AND AGREE TO BE BOUND BY ITS TERMS AND CONDITIONS. YOU ALSO AGREE THAT THIS AGREEMENT IS THE COMPLETE AND EXCLUSIVE STATEMENT OF THE AGREEMENT BETWEEN YOU AND THE COMPANY AND SUPERSEDES ALL PROPOSALS OR PRIOR AGREEMENTS, ORAL, OR WRITTEN, AND ANY OTHER COMMUNICATIONS BETWEEN YOU AND THE COMPANY OR ANY REPRESENTATIVE OF THE COMPANY RELATING TO THE SUBJECT MATTER OF THIS AGREEMENT.

Should you have any questions concerning this Agreement or if you wish to contact the Company for any reason, please contact in writing at the address below.

Robin Short
Prentice Hall PTR
One Lake Street
Upper Saddle River, New Jersey 07458

Andrew S. Tanenbaum

Architettura dei Calcolatori
un approccio strutturale

Quinta edizione

Pur conservando l'ormai nota e inconfondibile metafora che vede un calcolatore come una gerarchia di livelli, questa nuova edizione del best-seller di Andrew Tanenbaum presenta numerose e importanti novità, in linea con lo stato dell'arte dei più recenti sviluppi dell'industria dei computer. Gli esempi presentati riguardano l'Intel Pentium 4, il Sun Ultra-SPARC III e il processore Intel 8051, un chip oggi molto utilizzato in seguito alla crescente popolarità dei sistemi integrati. L'8051 rappresenta una scelta ampiamente diffusa grazie al costo estremamente basso, alla grande disponibilità di software e periferiche e all'esistenza di un gran numero di programmatori specializzati.
Inoltre, "a grande richiesta" è stata aggiunta un'interessante sezione, chiara e dettagliata, dedicata alla programmazione in linguaggio assemblativo, facendo riferimento a quello dell'8088, una versione ridotta del Pentium: anche in questo caso si tratta di una scelta dettata dalle esigenze didattiche, visto che la maggior parte degli studenti possiede, a casa, un computer con processore Pentium (sul quale, tra l'altro, è possibile eseguire programmi 8088).

L'Autore

Andrew S. Tanenbaum si è laureato al MIT, ha conseguito il Ph.D a Berkeley e da più di trent'anni è professore alla Vrije Universiteit di Amsterdam, dove ha insegnato sistemi operativi, reti e architetture dei calcolatori a migliaia di studenti. Con Prentice Hall ha pubblicato 15 best-seller, premiati con numerosi riconoscimenti da parte di ACM e altre associazioni professionali. È Fellow di ACM e IEEE.

Supplementi

Il **CD-ROM** allegato mette a disposizione un insieme di strumenti per facilitare l'apprendimento della programmazione assemblativa, tra cui un assemblatore 8088, un simulatore e un tracer.
Sul sito web **http://hpe.pearsoned.it** i docenti che adottano il testo potranno accedere alle slide in PowerPoint; contattando un rappresentante Pearson Education, è inoltre possibile richiedere l'Instructor's Manual (in inglese) con le soluzioni dei problemi.

ISBN 88-7192-271-9

PEARSON Education

Visitate il nostro sito web
http://hpe.pearsoned.it

€ 49,00

Garanzia limitata

LEGGERE L'ACCORDO PRIMA DI APRIRE LA CONFEZIONE DEL CD-ROM

Aprendo la confezione sigillata del CD-ROM dichiarate di accettare i termini e le condizioni riportate di seguito. Qualora decidiate di non accettare **non aprite la confezione**, ma restituitela sigillata.

Il software del CD-ROM è distribuito "AS IS" senza alcuna garanzia. Gli autori, gli sviluppatori del software e l'editore non forniscono alcuna garanzia, espressa o esplicita, rispetto ai programmi software, alla loro qualità, accuratezza o idoneità a scopi specifici. Gli autori, gli sviluppatori del software e l'editore non si assumono quindi alcuna responsabilità nei confronti di persone o enti rispetto a danni o perdite presunti o reali causati direttamente o indirettamente dai programmi contenuti nel CD-ROM, includendo, ma non limitandosi a, interruzione del servizio, perdita di dati, perdita di ore scolastiche, perdita di profitti o danni conseguenti dall'uso dei programmi. Il CD-ROM è sostituibile in caso di difetti di produzione.

Non sono consentiti reverse engineering, decompilazione e disassemblaggio del software. Non è consentito modificare o tradurre il software, né distribuirne copie senza il permesso scritto di Pearson Education Italia Srl.

Note per l'utilizzo del CD-ROM

Per visionare i contenuti del CD-ROM allegato, aprite il file index.htm con un browser Web.

Ricordate che la struttura della directory di questo CD-ROM segue la convenzione "otto-punto-tre" per i nomi di file e directory. Tenetelo presente nei casi in cui le istruzioni per l'installazione dell'Assembler 8088 e di Tracer, su tutte le piattaforme, si riferiscano ancora all'originaria struttura di directory. Per esempio, seguendo le istruzioni del sistema operativo che state utilizzando, dovrete digitare "8088_tra" e non "8088_tracer".

Contenuti del CD-ROM

- ***8088 Assembler* e *Tracer Toolkit***
  - Funzionano su ciascuna delle seguenti piattaforme: Microsoft Windows, piattaforme Intel/Linux (ovvero little endian) e piattaforme SPARC/Solaris (ovvero big endian).
- **Ambiente di simulazione Mic-1 MMV**
  - Scritto in Java™ e funzionante perciò su qualunque piattaforma Java.

Requisiti minimi di sistema

- Hardware e sistema operativo:

  Pentium II 450 MHz o superiore con Microsoft Windows Server 2003, XP Professional, XP Home Edition, XP Media Center Edition, XP Tablet PC Edition, 2000 Professional (è richiesto SP3 o successivo per l'installazione), 2000 Server (è richiesto SP3 o successivo per l'installazione), Windows ME, Windows 98 (1ª o 2ª edizione), oppure

  Piattaforme Intel/Linux (ovvero little endian): RedHat 9.0 oppure

  Macintosh; MacOS X 10.3 (solo Mic-1 MMV), oppure

  Piattaforme SPARC/Solaris (ovvero big endian): Solaris 8-10 (solo piattaforma SPARC)
- 10 MB di spazio sul disco rigido
- Lettore CD-ROM
- Browser Internet (Internet Explorer 6.0, Firefox 1.0, Mozilla 1.7.x, Safari 1.2
- Connessione a Internet (necessaria solo per effettuare il download di software ag namenti)
- Risoluzione minima del monitor: 800 x 600
- Java JRE 1.4 (richiesta solo se è installato il software Mic-1)
- Utilità Zip (richiesta solo se avete intenzione di modificare e ricompilare i file sor Mic-1).

SEREGNO
030001 015801
1

Biblioteca Civica
Ettore Pozzoli – Seregno
1030001015801
01/08/06
004 TAN